



SC140 DSP Core

Reference Manual

Revision 4.1, September 2005


This document contains information on a new product. Specifications and information herein are subject to change without notice.

(c) Freescale Semiconductor, Inc. 2005, All rights





LICENSOR is defined as Freescale Semiconductor, Inc. LICENSOR reserves the right to make changes without further notice to any products included and covered hereby. LICENSOR makes no warranty, representation or guarantee regarding the suitability of its products for any particular purpose, nor does LICENSOR assume any liability arising out of the application or use of any product or circuit, and specifically disclaims any and all liability, including without limitation incidental, consequential, reliance, exemplary, or any other similar such damages, by way of illustration but not limitation, such as, loss of profits and loss of business opportunity. "Typical" parameters which may be provided in LICENSOR data sheets and/or specifications can and do vary in different applications and actual performance may vary over time. All operating parameters, including "Typicals" must be validated for each customer application by customer's technical experts. LICENSOR does not convey any license under its patent rights nor the rights of others. LICENSOR products are not designed, intended, or authorized for use as components in systems intended for surgical implant into the body, or other applications intended to support life, or for any other application in which the failure of the LICENSOR product could create a situation where personal injury or death may occur. Should Buyer purchase or use LICENSOR products for any such unintended or unauthorized application, Buyer shall indemnify and hold LICENSOR and its officers, employees, subsidiaries, affiliates, and distributors harmless against all claims, cost, damages, and expenses, and reasonable attorney fees arising out of, directly or indirectly, any claim of personal injury or death associated with such unintended or unauthorized use, even if such claim alleges that LICENSOR was negligent regarding the design or manufacture of the part.

Freescale and  are registered trademarks of Freescale Semiconductor, Inc. Freescale, Inc. is an Equal Opportunity/Affirmative Action Employer.

All other tradenames, trademarks, and registered trademarks are the property of their respective owners.

Table of Contents

About This Book

Audience	xxi
Organization	xxi
Abbreviations	xxii
Revision History	xxiv

Chapter 1 Introduction

1.1	Target Markets	1-1
1.2	Architectural Differentiation.	1-2
1.3	Core Architecture Features	1-3
1.3.1	Typical System-On-Chip Configuration.	1-4
1.3.2	Variable Length Execution Set (VLES) Software Model	1-5

Chapter 2 Core Architecture

2.1	Architecture Overview	2-1
2.1.1	Data Arithmetic Logic Unit (DALU)	2-2
2.1.1.1	Data Register File	2-3
2.1.1.2	Multiply-Accumulate (MAC) Unit.	2-3
2.1.1.3	Bit-Field Unit (BFU)	2-3
2.1.1.4	Shifter/Limiters.	2-3
2.1.2	Address Generation Unit (AGU)	2-3
2.1.2.1	Stack Pointer Registers	2-4
2.1.2.2	Bit Mask Unit (BMU).	2-4
2.1.3	Program Sequencer Unit (PSEQ)	2-5
2.1.4	Enhanced On-Chip Emulator (EOnCE)	2-5
2.1.5	Instruction Set Accelerator Plug-in (ISAP) Interface.	2-5
2.1.6	Memory Interface	2-5
2.2	DALU	2-6
2.2.1	DALU Architecture	2-6
2.2.1.1	Data Registers (D0–D15)	2-8
2.2.1.2	Multiply-Accumulate (MAC) Unit.	2-10
2.2.1.3	Bit-Field Unit (BFU)	2-12
2.2.1.4	Data Shifter/Limiter	2-13
2.2.1.5	Scaling	2-14
2.2.1.6	Limiting	2-14
2.2.1.7	Scaling and Arithmetic Saturation Mode Interactions	2-16
2.2.2	DALU Arithmetic and Rounding	2-17



2.2.1	Data Representation	2-17
2.2.2.2	Data Formats	2-18
2.2.2.3	Multiplication	2-20
2.2.2.4	Division	2-20
2.2.2.5	Unsigned Arithmetic	2-20
2.2.2.6	Rounding Modes	2-21
2.2.2.7	Arithmetic Saturation Mode	2-25
2.2.2.8	Multi-Precision Arithmetic Support	2-26
2.2.2.9	Viterbi Decoding Support	2-30
2.3	Address Generation Unit	2-31
2.3.1	AGU Architecture	2-31
2.3.2	AGU Programming Model	2-34
2.3.2.1	Address Registers (R0–R15)	2-35
2.3.2.2	Stack Pointer Registers (NSP, ESP)	2-35
2.3.2.3	Offset Registers (N0–N3)	2-36
2.3.2.4	Base Address Registers (B0–B7)	2-36
2.3.2.5	Modifier Registers (M0–M3)	2-36
2.3.2.6	Modifier Control Register (MCTL)	2-37
2.3.3	Addressing Modes	2-38
2.3.3.1	Register Direct Modes	2-38
2.3.3.2	Address Register Indirect Modes	2-38
2.3.3.3	PC Relative Mode	2-40
2.3.3.4	Special Addressing Modes	2-41
2.3.3.5	Memory Access Width	2-42
2.3.3.6	Memory Access Misalignment	2-42
2.3.3.7	Addressing Modes Summary	2-43
2.3.4	Address Modifier Modes	2-45
2.3.4.1	Linear Addressing Mode	2-45
2.3.4.2	Reverse-carry Addressing Mode	2-45
2.3.4.3	Modulo Addressing Mode	2-45
2.3.4.4	Multiple Wrap-Around Modulo Addressing Mode	2-47
2.3.5	Arithmetic Instructions on Address Registers	2-48
2.3.6	Bit Mask Instructions	2-49
2.3.6.1	Bit Mask Test and Set (Semaphore Support) Instruction	2-50
2.3.6.2	Semaphore Hardware Implementation	2-51
2.3.7	Move Instructions	2-51
2.4	Memory Interface	2-55
2.4.1	SC140 Endian Support	2-56
2.4.1.1	SC140 Bus Structure	2-56
2.4.1.2	Memory Organization	2-57
2.4.1.3	Data Moves	2-58
2.4.1.4	Multi-Register Moves	2-60
2.4.1.5	Instruction Word Transfers	2-62
2.4.1.6	Memory Access Behavior in Big/Little Endian Modes	2-64



Chapter 3 Control Registers

3.1	Core Control Registers	3-1
3.1.1	Status Register (SR)	3-1
3.1.2	Exception and Mode Register (EMR)	3-7
3.1.2.1	Clearing EMR Bits	3-10
3.2	PLL and Clock Registers	3-10

Chapter 4 Emulation and Debug (EOnCE)

4.1	Debugging System	4-1
4.2	Overview of the Combined JTAG and EOnCE Interface	4-2
4.2.1	Cascading Multiple SC140 EOnCE Modules in a SoC	4-2
4.2.2	JTAG Scan Paths	4-3
4.2.3	Activating the EOnCE Through the JTAG Port	4-6
4.2.4	Enabling the EOnCE Module	4-6
4.2.5	DEBUG_REQUEST and ENABLE_EONCE Commands	4-7
4.2.6	Reading/Writing EOnCE Registers Through JTAG	4-7
4.3	Main Capabilities of the EOnCE Module	4-10
4.3.1	EOnCE Signals	4-10
4.3.2	EOnCE Dedicated Instructions	4-11
4.3.3	Debug State	4-11
4.3.4	Debug Exception	4-12
4.3.5	Executing an Instruction while in Debug State	4-12
4.3.6	Software Downloading	4-12
4.3.7	EOnCE Events	4-14
4.3.8	EOnCE Actions	4-15
4.3.9	Event and Action Summary	4-15
4.4	EOnCE Enabling and Power Considerations	4-16
4.5	EOnCE Module Internal Architecture	4-16
4.5.1	EOnCE Controller	4-16
4.5.2	Event Counter	4-18
4.5.3	Event Detection Unit (EDU)	4-20
4.5.3.1	Address Event Detection Channel (EDCA)	4-22
4.5.3.2	Data Event Detection Channel (EDCD)	4-24
4.5.3.3	Optional External Event Detection Address Channels	4-25
4.5.4	Event Selector (ES)	4-25
4.5.5	Trace Unit	4-26
4.5.5.1	Change of Flow and Interrupt Tracing	4-28
4.5.5.2	Writing to the Trace Buffer	4-29
4.5.5.3	Reading the Trace Buffer (TB_BUFF)	4-29
4.5.5.4	Trace Unit Programming Model	4-29
4.6	EOnCE Register Addressing	4-30
4.6.1	Reading or Writing EOnCE Registers Using Core Software	4-33
4.6.2	Real-Time JTAG Access	4-33
4.6.3	Real-Time Data Transfer	4-34



5.4	General EOnCE Register Issues	4-34
4.7	EOnCE Controller Registers.	4-36
4.7.1	EOnCE Command Register (ECR).	4-36
4.7.2	EOnCE Status Register (ESR)	4-37
4.7.3	EOnCE Monitor and Control Register (EMCR).	4-41
4.7.4	EOnCE Receive Register (ERCV)	4-43
4.7.5	EOnCE Transmit Register (ETRSMT).	4-43
4.7.6	EE Signals.	4-44
4.7.6.1	EE Signals as Outputs.	4-44
4.7.6.2	EE Signals as Inputs	4-45
4.7.6.3	EE Signals Control Register (EE_CTRL)	4-45
4.7.7	Core Command Register (CORE_CMD)	4-48
4.7.8	PC of the Exception Execution Set (PC_EXCP)	4-49
4.7.9	PC of the Next Execution Set (PC_NEXT)	4-49
4.7.10	PC of Last Execution Set (PC_LAST)	4-49
4.7.11	PC Breakpoint Detection Register (PC_DETECT)	4-49
4.8	Event Counter Registers	4-50
4.8.1	Event Counter Control Register (ECNT_CTRL)	4-50
4.8.2	Event Counter Value Register (ECNT_VAL)	4-52
4.8.3	Extension Counter Value Register (ECNT_EXT)	4-53
4.8.4	EC Signals.	4-53
4.9	Event Detection Unit (EDU) Channels and Registers	4-54
4.9.1	Address Event Detection Channel (EDCA)	4-54
4.9.1.1	EDCA Control Registers (EDCAi_CTRL).	4-54
4.9.1.2	EDCA Reference Value Registers A and B (EDCAi_REFA, EDCAi_REFB)	4-57
4.9.1.3	EDCA Mask Register (EDCAi_MASK)	4-57
4.9.2	Data Event Detection Channel (EDCD)	4-58
4.9.2.1	EDCD Control Register (EDCD_CTRL)	4-58
4.9.2.2	EDCD Reference Value Register (EDCD_REF)	4-61
4.9.2.3	EDCD Mask Register (EDCD_MASK)	4-61
4.10	Event Selector (ES) Registers.	4-61
4.10.1	Event Selector Control Register (ESEL_CTRL)	4-61
4.10.2	Event Selector Mask Debug State Register (ESEL_DM)	4-63
4.10.3	Event Selector Mask Debug Exception Register (ESEL_DI)	4-64
4.10.4	Event Selector Mask Enable Trace Register (ESEL_ETB)	4-64
4.10.5	Event Selector Mask Disable Trace Register (ESEL_DTB)	4-65
4.11	Trace Unit Registers	4-65
4.11.1	Trace Buffer Control Register (TB_CTRL)	4-65
4.11.2	Trace Buffer Read Pointer Register (TB_RD)	4-69
4.11.3	Trace Buffer Write Pointer Register (TB_WR)	4-69
4.11.4	Trace Buffer Register (TB_BUFF).	4-69

Chapter 5 Program Control

5.1	Pipeline	5-1
5.1.1	Instruction Pipeline Stages	5-2
5.1.1.1	Instruction Pre-Fetch and Fetch	5-4
5.1.1.2	Instruction Dispatch	5-4
5.1.1.3	Address Generation	5-4
5.1.1.4	Execution	5-5
5.2	Instruction Grouping	5-5
5.2.1	Grouping Types	5-6
5.2.1.1	Serial Grouping	5-7
5.2.1.2	Prefix Grouping	5-7
5.2.2	Prefix Types	5-8
5.2.2.1	Two-Word Prefix	5-8
5.2.2.2	One-Word Low Register Prefix	5-9
5.2.3	Conditional Execution	5-9
5.2.4	Prefix Selection Algorithm	5-10
5.2.5	Instruction Reordering Within an Execution Set	5-12
5.3	Instruction Timing	5-14
5.3.1	Sequential Instruction Timing	5-15
5.3.1.1	DALU Instruction Timing	5-16
5.3.1.2	Move Instruction Timing	5-16
5.3.1.3	Bit Mask Instruction Timing	5-16
5.3.2	Change-Of-Flow Instruction Timing	5-17
5.3.2.1	Direct, PC-Relative, and Conditional COF	5-18
5.3.2.2	Delayed COF	5-19
5.3.2.3	COF Execution Cycles	5-19
5.3.3	Memory Access Timing	5-21
5.3.3.1	Memory Access Examples	5-22
5.3.3.2	Implicit Push/Pop Memory Timing	5-24
5.3.3.3	Memory Stall Conditions	5-24
5.4	Hardware Loops	5-25
5.4.1	Loop Programming Model	5-25
5.4.1.1	Loop Start Address Registers (SAn)	5-25
5.4.1.2	Loop Counter Registers (LCn)	5-26
5.4.1.3	Status Register (SR) Loop Flag Bits	5-26
5.4.2	Loop Notation and Encoding	5-26
5.4.3	Loop Initiation and Execution	5-27
5.4.4	Loop Nesting	5-28
5.4.5	Loop Iteration and Termination	5-28
5.4.6	Loop Control Instructions	5-29
5.4.7	Loop Timing	5-32
5.5	Stack Support	5-32
5.5.1	SC140 Single Stack Memory Use	5-32
5.5.2	SC140 Dual Stack Memory Use	5-33
5.5.3	Stack Support Instructions	5-34
5.5.4	Shadow Stack Pointer Registers	5-35



5.5	Fast Return from Subroutines	5-36
5.6	Working Modes	5-37
5.6.1	Normal Working Mode.	5-37
5.6.2	Exception Working Mode.	5-37
5.6.3	Typical Working Mode Usage Scenarios	5-38
5.6.3.1	Dual-stack RTOS	5-38
5.6.3.2	Single-stack RTOS	5-39
5.6.4	Working Mode Transitions	5-39
5.6.4.1	From Exception to Normal mode	5-39
5.6.4.2	From Normal to Exception mode	5-39
5.7	Processing States.	5-41
5.7.1	Processing State Change Instructions	5-41
5.7.2	Processing State Transitions	5-42
5.7.3	Execution State	5-43
5.7.4	Reset Processing State	5-43
5.7.5	Debug State.	5-44
5.7.6	Wait Processing State	5-44
5.7.7	Stop Processing State	5-45
5.8	Exception Processing	5-46
5.8.1	Interrupt Vector Address	5-48
5.8.1.1	Vector Base Address Register	5-48
5.8.1.2	Programming Exception Routine Addresses	5-48
5.8.2	Return From Exception Instructions.	5-49
5.8.3	Maskable Interrupts	5-50
5.8.3.1	Interrupt Priority Level	5-50
5.8.3.2	Controlling All Interrupt Sources	5-50
5.8.4	Non-Maskable Interrupts (NMI).	5-50
5.8.5	Internal Exceptions	5-50
5.8.5.1	Illegal Exception	5-51
5.8.5.2	DALU Overflow	5-52
5.8.5.3	TRAP Exception	5-52
5.8.5.4	Debug Exception.	5-52
5.8.6	Exception Interface to the Pipeline.	5-52
5.8.6.1	Exception Routine Fetch.	5-52
5.8.6.2	Exception Mode Execution.	5-53
5.8.7	Exception Timing	5-53

Chapter 6 Instruction Set Accelerator Plug-In

6.1	Introduction.	6-57
6.2	ISAP - SC140 Schematic Connection	6-58
6.2.1	Single ISAP.	6-58
6.2.2	Multiple ISAP	6-59
6.3	ISAP instructions and instruction encoding	6-60
6.4	ISAP Memory Access.	6-60
6.5	ISAP-core register transfers	6-61
6.6	Immediate Data Transfer to ISAP registers	6-62



7	Core Assembly Syntax with an ISAP	6-63
6.7.1	Identification of ISAP instructions	6-63
6.7.1.1	Working with One ISAP	6-63
6.7.1.2	Working with Multiple ISAPs	6-64
6.7.2	An Example of the Definition Flexibility of an ISAP	6-65
6.7.3	Conditional Execution	6-66
6.8	Programming Rules	6-67
6.8.1	ISAP Functions that Interact With the Core	6-67
6.8.2	Grouping rules for explicit ISAP instructions	6-68
6.8.3	Rules for implicit AGU instructions	6-68
6.8.4	Sequencing rules for T bit update	6-69

Chapter 7 Programming Rules

7.1	VLES Sequencing Semantics	7-1
7.2	VLES Grouping Semantics	7-1
7.3	SC140 Pipeline Exposure	7-3
7.4	Programming Rule Notation	7-3
7.4.1	Grouping Rules	7-3
7.4.1.1	Prefix Instructions	7-3
7.4.1.2	Conditional Subgroups	7-3
7.4.1.3	Assembler Reordering	7-3
7.4.2	Sequencing Rules	7-4
7.4.2.1	Cycle Counts	7-4
7.4.2.2	Conditional Execution	7-4
7.4.2.3	Simulator Execution Counts	7-4
7.4.3	Register Read/Write	7-4
7.4.3.1	Register Names	7-4
7.4.3.2	B Register Aliasing	7-5
7.4.4	Status Bit Updates	7-5
7.4.5	Instruction Words	7-5
7.4.6	MOVE-like Instructions	7-5
7.4.6.1	Address/Data Operands	7-5
7.4.7	AGU Arithmetic Instructions	7-6
7.4.8	Change-Of-Flow Destinations	7-6
7.4.8.1	COF Instructions	7-6
7.4.9	Delayed COF Instructions	7-6
7.4.9.1	Delay Slot	7-6
7.4.10	Hardware Loops	7-7
7.4.10.1	Enabled Loop	7-7
7.4.10.2	Enveloping Loop	7-7
7.5	Static Programming Rules	7-7
7.5.1	Hardware Loop Detection	7-7
7.5.2	General Grouping Rules	7-8
7.5.3	Prefix Grouping Rules	7-11
7.5.4	AGU Rules	7-16
7.5.5	Delayed COF Rules	7-19



5.6	Status Bit Rules	7-22
7.5.7	Loop Nesting Rules	7-28
7.5.8	Loop LA Rules	7-31
7.5.9	Loop Sequencing Rules	7-33
7.5.10	Loop COF Rules	7-36
7.5.11	General Looping Rules	7-40
7.6	Dynamic Programming Rules	7-41
7.6.1	AGU Dynamic Rules	7-41
7.6.2	Memory Access Rules	7-42
7.6.3	RAS Rules	7-43
7.6.4	Loop Rules	7-43
7.6.5	Rule Detection Across COF Boundaries	7-44
7.6.5.1	Cycle-Based COF Rules	7-44
7.6.5.2	VLES-Based COF Rules	7-45
7.6.6	Rule Detection Across Exception Boundaries	7-46
7.7	Programming Guidelines	7-48
7.7.1	Rules Not Detected Across COF Boundaries	7-49
7.7.2	Good Programming Practices	7-49
7.7.2.1	Source Code Practices	7-49
7.7.2.2	Binary Code Practices	7-50
7.7.2.3	Software Development Practices	7-51
7.8	LPMARK Rules	7-51
7.8.1	LPMARK Instruction Type	7-51
7.8.2	Static Programming Rules	7-52
7.8.2.1	General Grouping Rules	7-52
7.8.2.2	Prefix Grouping Rules	7-52
7.8.3	Dynamic Programming Rules	7-52
7.8.3.1	LPMARK Notation	7-52
7.8.3.2	Loop Nesting Rules	7-53
7.8.3.3	Loop LA Rules	7-53
7.8.3.4	Loop Sequencing Rules	7-55
7.8.3.5	Loop COF Rules	7-56
7.8.3.6	General Looping Rules	7-59
7.8.3.7	Rule Detection Across Exception Boundaries	7-59
7.8.4	LPMARK Programming Guidelines	7-59
7.9	NOP Definition	7-60
7.9.1	Grouping Examples	7-61

Appendix A SC140 DSP Core Instruction Set

A.1	Introduction	A-1
A.1.1	Conventions	A-2
A.1.1.1	Brackets as ISAP indicators	A-4
A.1.1.2	Brackets as address indicators	A-4
A.1.2	Addressing Mode Notation	A-5
A.1.3	Data Representation in Memory for the Examples	A-6
A.1.4	Encoding Notation	A-6



- 1.5 Prefix Word Encoding A-7
- A.1.5.1 One-Word Low Register Prefix A-8
- A.1.5.2 Two-Word Prefix A-9
- A.1.6 Instruction Types A-12
- A.1.6.1 Instruction Sub-types A-12
- A.2 Instructions A-19
- A.2.1 Instruction Definition Layout A-19

Appendix B
StarCore Registry

- B.1 Using the StarCore Registry B-1



List of Figures

1-1	Block Diagram of a Typical SoC Configuration with the SC140 Core	1-5
2-1	Block Diagram of the SC140 Core	2-2
2-2	DALU Architecture	2-6
2-3	DALU Data Representations	2-18
2-4	Fractional and Integer Multiplication	2-20
2-5	Convergent Rounding (No Scaling)	2-22
2-6	Two's Complement Rounding (No Scaling)	2-24
2-7	DMAC Implementation	2-26
2-8	Fractional Double-Precision Multiplication	2-27
2-9	Fractional Mixed-Precision Multiplication.	2-28
2-10	Signed Integer Double-Precision Multiplication	2-29
2-11	Unsigned Integer Double-Precision Multiplication	2-30
2-12	AGU Block Diagram	2-32
2-13	AGU Programming Model	2-34
2-14	Modifier Control Register (MCTL) Format	2-37
2-15	Modulo Addressing Example	2-46
2-16	Integer Move Instructions	2-53
2-17	Fractional Move Instructions	2-54
2-18	Bit Allocation in MOVE.L D0.e:D1.e	2-55
2-19	Endian Example	2-56
2-20	Basic Connection between SC140 Core and Memory	2-57
2-21	Memory Organization of Big and Little Endian Mode.	2-57
2-22	Data Transfer in Big and Little Endian Modes.	2-59
2-23	Multi-Register Transfer in Big and Little Endian Modes.	2-61
2-24	Program Memory Organization in Big and Little Endian Modes	2-62
2-25	Instruction Moves in Big and Little Endian Modes	2-63
3-1	Status Register -SR	3-2
3-2	Exception and Mode Register (EMR)	3-7
4-1	JTAG and EOnCE Multi-core Interconnection	4-3
4-2	TAP Controller State Machine	4-5
4-3	Cascading Multiple EOnCE Modules.	4-7
4-4	Reading and Writing EOnCE Registers Via JTAG	4-8
4-5	Accessing EOnCE registers through JTAG	4-9
4-6	Typical Debugging System.	4-10

7	Software Downloading	4-13
4-8	EOnCE Controller Block Diagram	4-17
4-9	Event Counter Block Diagram	4-19
4-10	Event Detection Unit Block Diagram	4-21
4-11	EDCA Block Diagram	4-22
4-12	EDCD Block Diagram	4-24
4-13	Event Selector Block Diagram	4-26
4-14	Trace Unit Block Diagram	4-28
4-15	EOnCE Command Register (ECR).	4-36
4-16	EOnCE Status Register (ESR)	4-38
4-17	EOnCE Monitor and Control Register (EMCR).	4-41
4-18	EE Signals Control Register (EE_CTRL)	4-45
4-19	Injected Instruction Format.	4-48
4-20	Event Counter Register (ECNT_CTRL).	4-51
4-21	EDCA Control Register (EDCAi_CTRL)	4-54
4-22	EDCD Control Register (EDCD_CTRL).	4-58
4-23	Event Selector Control Register (ESEL_CTRL)	4-62
4-24	Event Selector Mask Debug State (ESEL_DM).	4-63
4-25	Event Selector Mask Debug Exception (ESEL_DI).	4-64
4-26	Event Selector Mask Enable Trace (ESEL_ETB)	4-64
4-27	Event Selector Mask Disable Trace (ESEL_DTB).	4-65
4-28	Trace Buffer Control Register (TB_CTRL)	4-67
5-1	Instruction Pipeline Stages	5-2
5-2	Instruction Grouping Methods	5-6
5-3	Low Register Prefix Selection Algorithm	5-11
5-4	Hardware Loop Programming Model.	5-25
5-5	Loop Nesting.	5-28
5-6	SC140 Memory Use with a Single Stack Pointer.	5-32
5-7	SC140 Memory Use with Dual Stack Pointers.	5-33
5-8	Working mode Transitions - Unprotected Dual-stack RTOS.	5-38
5-9	Working mode Transitions - Unprotected Single-stack RTOS	5-39
5-10	Core State Diagram.	5-42
5-11	Core-PIC Interface	5-47
5-12	Flowchart for Exception Timing.	5-55
6-1	Core to Single ISAP Connection Schematic.	6-58
6-2	Core to Multiple ISAP Connection Schematic	6-59

List of Tables

2-1	DALU Programming Model	2-7
2-2	Write to Data Registers	2-9
2-3	Read from Data Registers	2-9
2-4	Data Registers Access Width	2-10
2-5	DALU Arithmetic Instructions (MAC)	2-10
2-6	DALU Logical Instructions (BFU)	2-13
2-7	Scaling Example	2-14
2-8	Ln Bit Calculation	2-15
2-9	Limiting Example	2-16
2-10	Scaling and Limiting Interactions	2-16
2-11	Saturation and Rounding Interactions	2-17
2-12	Two's Complement Word Representations	2-19
2-13	Rounding Position in Relation to Scaling Mode	2-21
2-14	Arithmetic Saturation Example	2-25
2-15	Fractional Signed and Unsigned Two's Complement Multiplication	2-26
2-16	Integer Signed and Unsigned Two's Complement Multiplication	2-28
2-17	Address Modifier (AM) Bits	2-37
2-18	Access Width Support for Address and Register Update Calculations	2-42
2-19	Memory Address Alignment	2-43
2-20	Addressing Modes Summary	2-43
2-21	Modulo Register Values for Modulo Addressing Mode	2-47
2-22	Modulo Register Values for Wrap-Around Modulo Addressing Mode	2-48
2-23	AGU Arithmetic Instructions	2-48
2-24	AGU Bit Mask Instructions (BMU)	2-50
2-25	AGU Move Instructions	2-52
2-26	Data Representation in Memory	2-58
2-27	Move Instructions in Big and Little Endian Modes	2-64
2-28	Stack Support Instructions in Big and Little Endian Modes	2-67
2-29	Bit Mask Instructions in Big and Little Endian Modes	2-67
2-31	Control Instructions in Big and Little Endian Modes	2-68
2-30	Non-Loop Change-of-Flow Instructions in Big and Little Endian Modes	2-68
3-1	Status Register Description	3-2
3-2	EMR Description	3-8
4-1	JTAG Interface Signal Descriptions	4-2



2	JTAG Instructions	4-3
4-3	JTAG Scan Paths	4-5
4-4	EOnCE Event Types	4-14
4-5	EOnCE Event and Action Summary	4-15
4-6	EOnCE Controller Register Set	4-17
4-7	Event Counter Register Set	4-19
4-8	EDCA Register Set	4-23
4-9	EDCD Register Set	4-24
4-10	Event Selector Register Set	4-26
4-11	Trace Buffer Register Set	4-30
4-12	EOnCE Register Addressing Offsets	4-31
4-13	ECR Description	4-36
4-14	ESR Description	4-38
4-15	EMCR Description	4-41
4-16	EE_CTRL Description	4-46
4-17	Length Control Bits	4-48
4-18	ECNT_CTRL Description	4-51
4-19	EDCA_CTRL Description	4-54
4-20	EDCD_CTRL Description	4-58
4-21	ESEL_CTRL Description	4-62
4-22	Allowed tracing mode combinations	4-66
4-23	TB_CTRL Description	4-67
5-1	Pipeline Example	5-3
5-2	Pipeline Stages Overview	5-3
5-3	Prefix Instructions	5-9
5-4	Conditional IFc Syntax	5-9
5-5	Instruction Categories Timing Summary	5-15
5-6	Non-Loop Change-of-Flow Instructions	5-17
5-7	Loop Change-Of-Flow Instructions	5-18
5-8	Number of Cycles Needed by Change-of-Flow Instructions	5-20
5-9	LPMARKA and LPMARKB Bits in Short and Long Loops	5-27
5-10	Loop Control Instructions	5-29
5-11	Stack Push/Pop Instructions	5-34
5-12	Even and Odd Registers	5-34
5-13	Stack Memory Map	5-35
5-14	Stack Move Instructions	5-35
5-15	Working Modes	5-37
5-16	Processing State Change Instructions	5-41
5-17	Processing State Transitions	5-43



18	Exit Wait Processing State due to an Interrupt or NMI	5-45
5-19	Exception Vector Address Table	5-49
5-20	Exception Pipeline	5-53
5-21	Pipeline Example	5-56
6-1	ISAP Encoding Fields.	6-60
A-1	Instruction Conventions	A-2
A-2	Operations Syntax.	A-3
A-3	Register Abbreviations	A-3
A-4	Assembler Syntax	A-4
A-5	Addressing Mode Notation for the EA Operand	A-5
A-6	Addressing Mode Notation for the ea Operand	A-5
A-7	DALU Arithmetic Instructions (MAC)	A-13
A-8	DALU Logical Instructions (BFU).	A-14
A-9	AGU Arithmetic Instructions	A-15
A-10	AGU Move Instructions	A-15
A-11	AGU Stack Support Instructions	A-16
A-12	AGU Bit-Mask Instructions (BMU).	A-17
A-13	AGU Non-Loop Change-of-Flow Instructions.	A-17
A-14	AGU Loop Control (Including Loop COF) Instructions	A-18
A-15	AGU Program Control Instructions	A-18
A-16	Prefix Instructions.	A-18
A-17	Combinations of LPMARKx Use.	A-221
B-1	SCID Assignments	B-2



List of Examples

3-1	Clearing an EMR Bit	3-10
5-1	Four SC140 Instructions in an Execution Set.	5-5
5-2	Grouping Six SC140 Instructions in an Execution Set.	5-5
5-3	Execution Set with Three One-word and Two Two-word Instructions	5-13
5-4	Conditional VLES Having Two Subgroups	5-13
5-5	Set of 2 Two-word Instructions Requiring a NOP	5-13
5-6	Delayed Change-of-Flow and Its Delay Slot	5-17
5-7	Subroutine Call Timing	5-20
5-8	Parallel Execution of Two Move Instructions	5-23
5-9	Execution Set Containing a Bit Mask and a Move Instruction.	5-23
5-10	Execution Set Containing One Bit Mask Instruction	5-23
5-11	Execution Set Containing a Bit Mask and a Pop Instruction	5-24
5-12	Long Loop.	5-30
5-13	Long Loop Disassembly	5-30
5-14	Short Loop, Two Execution Sets	5-30
5-15	Short Loop, One Execution Set	5-31
5-16	Nested Loop	5-31
5-17	Basic Exception Timing	5-53
6-1	ISAP memory access	6-61
6-2	ISAP-Core register transfers.	6-62
6-3	ISAP-Core register transfers.	6-62
6-4	Single ISAP coding.	6-63
6-5	Multiple ISAP coding.	6-65
6-6	Conditional Execution Example	6-66
6-7	Conditional Execution Example	6-66
6-8	MOVE rules with an implicit MOVE instruction from ISAP	6-68
7-1	B Register Aliasing.	7-5
7-2	Delayed COF Instructions	7-6
7-3	VLES Word Count Exceeds Eight	7-8
7-4	Too Many AGU Instructions	7-8
7-5	Duplicate PC Destinations	7-9
7-6	Duplicate Address Pointer Register Destinations.	7-9



7	Duplicate Stack Pointer Destinations	7-9
7-8	Duplicate Register Destinations	7-10
7-9	Duplicate SR/EMR Register Destinations	7-10
7-10	Duplicate Status Bit Destinations	7-10
7-11	Dual Stack Pointer Destination Exception	7-10
7-12	Mutually Exclusive Register Destination Exception	7-11
7-13	Mutually Exclusive Status Bit Destination Exception	7-11
7-14	Multiple C, S and DOVF Status Bit Destination Exception.	7-11
7-15	DALU Register Use Exceeds Four Times	7-11
7-16	VLES Extension Words Exceed Two.	7-12
7-17	Two-Word Instructions Exceed Two	7-12
7-18	VLES Has Mutually Exclusive Instructions.	7-13
7-19	RTE Uses Both AAU	7-13
7-20	Data Source Use of Nn and Mn Registers	7-14
7-21	IFc Having Two Subgroups	7-14
7-22	IFA Subgroup Must Be Last Instructions.	7-14
7-23	Core AGU instructions on same VLES as ISAP instructions	7-15
7-24	ISAP instructions in same IFc group	7-15
7-25	MCTL Write to R0-R7 Use	7-16
7-26	Rn, Nn, Mn Write to AGU Use	7-17
7-27	Rn or Nn Write to MOVE-like Use	7-18
7-28	LCn Write to MOVE-like Use	7-18
7-29	NMID Update to EMR Read	7-19
7-30	Instructions in a Delay Slot.	7-19
7-31	Instructions in a RTED Delay Slot.	7-20
7-32	RTE/D with SR Updates	7-20
7-33	PC Read in a Return Delay Slot	7-21
7-34	SR Write with a Subroutine Call	7-21
7-35	SR Write in BSRD or JSRD Delay Slot.	7-21
7-36	SP Use in Return Delay Slots	7-21
7-37	SR Read in a CONTD Delay Slot.	7-22
7-38	EMR Use in Return Delay Slots.	7-22
7-39	T Bit Update to IFT/IFF AGU Use.	7-22
7-40	T Bit Update by ISAP and COF	7-23
7-41	T Bit Update by ISAP and MOVET/MOVEF	7-23
7-42	T Bit Update by ISAP and IFT/IFF	7-23
7-43	SR Write to SR Status Bit Use	7-25

44	SR Write to SR Status Bit Update	7-26
7-45	DOVF Update to SR Read or Write	7-27
7-46	DOVF Update grouped with Move-like SR updates	7-27
7-47	Status Bit Update with SR Read	7-28
7-48	Nested Loops with the Same LA	7-28
7-49	Nested Loops with Ordered Index	7-29
7-50	Nested DOENn/DOENSHn Instructions	7-29
7-51	DOENn instruction following DOENSHn Instruction	7-30
7-52	LOOPEND between DOEN and LOOPEND	7-30
7-53	Changing a loop type	7-30
7-54	Instructions at the End of Long Loops	7-31
7-55	LCn Write at the End of Long Loop n	7-31
7-56	Instructions in Short Loops	7-32
7-57	Short Loop LA at the End of a Long Loop	7-32
7-58	LCn Write to SKIPLS Instruction	7-33
7-59	LCn Write at the End of Long Loop n	7-33
7-60	LCn Write at the Start of Short Loop n	7-34
7-61	LCn Write to CONT/D Instruction	7-34
7-62	SAn Write at the End of Long Loop n	7-35
7-63	SAn Write to CONT/D Instruction	7-35
7-64	LCn Read at the Start of Short Loop n	7-35
7-65	COF Destination to Loop Delay Slots	7-36
7-66	COF Instructions at LA-2 of a Long Loop	7-36
7-67	Bc/Jc at SA-1 of a Short Loop	7-36
7-68	Bc/Jc at LA-3 of a Long Loop	7-37
7-69	Loop COF Destination in the Same Loop	7-38
7-70	Loop COF at End of Nested Long Loops	7-39
7-71	Subroutine Call to End of Loops	7-39
7-72	Delayed COF at LA-3 of a Long Loop	7-40
7-73	Delayed COF at SA-1 of a Short Loop	7-40
7-74	SR Read to LA of Any Long Loop	7-40
7-75	SR Read to SA of Any Short Loop	7-40
7-76	Enabling Short and Long Loops	7-41
7-77	Bn, Mn Write to AGU Use	7-41
7-78	Multiple Memory Writes to the Same Location	7-42
7-79	Pre-Calculated Memory Accesses to the Same Location	7-42
7-80	Memory Write to Stack in a Return Delay Slot	7-42

81	Illegal use of RAS value	7-43
7-82	SR.2 Across a COF Boundary	7-44
7-83	A.2 from a Delay Slot to a COF Destination	7-44
7-84	Set condition during a COF, and use it at the destination (T.1)	7-45
7-85	EMR access at the start of an exception	7-46
7-86	MCTL Write to R0-R7 Use	7-47
7-87	Invalid COF Destination Cannot be Detected	7-48
7-88	COF Destination in the Middle of a VLES.	7-48
7-89	COF Destination in a Delay Slot	7-48
7-90	LFn Enabled During Loop Body n	7-49
7-91	LFn Enabled at LPA or LPB.	7-53
7-92	Instructions at the End of Long Loops	7-53
7-93	Active LCn Write at the End of Long Loops	7-54
7-94	Instructions in Short Loops.	7-54
7-95	Active LCn Write at the Start of a Loop.	7-55
7-96	Active SAn Write at the End of Long Loops	7-55
7-97	Active LCn Read at the Start of a Loop	7-56
7-98	COF Instructions at LPB of a Long Loop.	7-57
7-99	Bc/Jc at the Start of a Loop.	7-57
7-100	Loop COF at End of Nested Long Loops.	7-58
7-101	Subroutine Call to End of Loops	7-58
7-102	Delay Slot at LPA or LPB of a Loop	7-59
7-103	SR Read to LPA or LPB of a Loop	7-59
7-104	COF Destination to Loop Delay Slots	7-60

This manual provides reference information for the StarCore SC140 digital signal processor (DSP) core. Specifically, this book describes the instruction set architecture and programming model for the SC140 core as well as corresponding register details, debug capabilities, and programming rules.

An appendix provides a detailed instruction reference for the SC140 instruction set, describing the operation, mnemonics, instruction fields, and encoding for each instruction. Instruction examples are also provided.

The resulting system-on-chip devices designed around the SC140 core will usually include additional functional blocks such as on-chip memory, an external memory interface, peripheral accelerators, and coprocessor devices. The specification of these functional blocks is customer-specific as well as application-specific. Therefore, this information is not covered in this manual.

Audience

This manual is intended for systems software developers, hardware designers, and application developers.

Organization

This book is organized into six chapters and one appendix as follows:

- [Chapter 1, “Introduction”](#), describes key features of the SC140 architecture. This chapter also illustrates a typical system using the SC140 core.
- [Chapter 2, “Core Architecture”](#), describes the main functional blocks and data paths of the SC140 core.
- [Chapter 3, “Control Registers”](#), details the core’s control registers.
- [Chapter 4, “Emulation and Debug \(EOnCE\)”](#), describes the hardware debug capabilities of the core.
- [Chapter 5, “Program Control”](#), details program control features such as the pipeline, instruction grouping, instruction timing, hardware loops, stack support, processing states, protection model, and exception processing.
- [Chapter 6, “Instruction Set Accelerator Plug-In”](#), describes how the SC140 core and SW developer can work with a an Instruction Set Accelerator Plug-In.
- [Chapter 7, “Programming Rules”](#), details the VLES semantics, static programming rules, dynamic programming rules, and programming guidelines for correct code construction.
- [Appendix A, “SC140 DSP Core Instruction Set,”](#) references the SC140 instruction set.
- [Appendix B, “StarCore Registry,”](#) shows how to access the core version

Abbreviations

The abbreviations used in this manual are listed below:

Table 1. Abbreviations

Abbreviation	Description
AAU	Address arithmetic unit
ADM	Application development module
AGU	Address generation unit
ALU	Arithmetic logic unit
Bn	AGU base address register n
BFU	Bit-field unit
BMU	Bit mask unit
DALU	Data arithmetic and logic unit
DSP	Digital signal processor
ECR	EOnCE control register
EDU	Event detection unit, with respect to the EOnCE
EE	EOnCE event pins
EMCR	EOnCE monitor and control register
EMR	Exception and mode register
EOnCE	Enhanced on-chip emulator
ERCV	EOnCE receive register
ES	Event selector, with respect to the EOnCE
ESP	Exception mode stack pointer
ESR	EOnCE status register
ETRSMT	EOnCE transmit register
EXT	Extension portion of a data register
FC	Fetch counter
FIFO	First-in first-out
FFT	Fast Fourier transform
HP	High portion of a data register
IPL	Interrupt priority level
ISAP	Instruction Set Accelerator Plug-in

Table 1. Abbreviations (Continued)

Abbreviation	Description
ISR	Interrupt service routine
JTAG	Joint test action group
LA	Last address
LCn	Loop counter register n
Ln	Limit tag bit n
LP	Low portion of a data register
LSB	Least significant bits
LSP	Least significant portion
Mn	AGU modifier register
MAC	Multiply-accumulate
MCTL	Modifier control register
MIPS	Million instructions per second
MMACS	Million multiply and accumulate operations per second
MSB	Most significant bits
MSP	Most significant portion
Nn	AGU offset register n
NMI	Non-maskable interrupt
NSP	Normal mode stack pointer
OS	Operating system
PAB	Program address bus
PAG	Program address generator
PC	Program counter register
PCU	Program control unit
PDB	Program data bus
PDU	Program dispatch unit
PIC	Programmable interrupt controller
PLL	Phase locked loop
PSEQ	Program sequencer unit
Rn	AGU address register n

Table 1. Abbreviations (Continued)

Abbreviation	Description
RAS	Return address register
RTOS	Real-time operating system
SAn	Start address register n
SF	Signed fractional
SI	Signed integer
SM	Saturation mode
SoC	System-on-chip
SP	Stack pointer
SR	Status register
T	True bit
UI	Unsigned integer
VBA	Interrupt vector base address register
VLES	Variable length execution set instruction grouping
XABA	Data memory address bus A
XABB	Data memory address bus B
XDBA	Data memory data bus A
XDBB	Data memory data bus B

Revision History

Table 2. Revision History

Revision	Date	Description
4.0	31 Aug, 2004	Fourth release of SC140
4.1	20 Sep, 2005	Misc. corrections (restored missing IADDNC.W instruction)

Chapter 1

Introduction

The StarCore SC140 digital signal processing (DSP) core, a new member of the SC100 architecture, addresses key market needs of next-generation DSP applications. It is especially suited for wireline and wireless communications, including infrastructure and subscriber communications. It is a flexible programmable DSP core which enables the emergence of computational-intensive communication applications by providing exceptional performance, low power consumption, efficient compilability, and compact code density. The SC140 core efficiently deploys a variable-length execution set (VLES) execution model which utilizes maximum parallelism by allowing multiple address generation and data arithmetic logic units to execute multiple instructions in a single clock cycle.

This chapter describes key features of the SC140 core architecture.

1.1 Target Markets

The design of the SC140 architecture aims to provide a DSP software platform that fulfills the constantly increasing computational requirements of DSP applications due to:

- New communication standards and services
- Wideband channels and data rates
- New user interfaces and media

Currently, software-configurable wireless terminals are already required to accommodate multiple air interfaces and frequency bands for cellular phones, PCs, paging devices, cordless phones, wireless LAN systems, and modems. In addition, multiple voice, messaging, internet, and video services must also be supported. These terminals must be flexible and upgradable so that they can be personalized for each user (such as permitting the dynamic download of applets). Finally, these terminals must be able to process baseband data using software to implement a range of functions previously carried out by hardware.

Target markets for the SC140 architecture include:

- Wireless software configurable handset terminals (radios)
- Third generation wireless handset systems with wideband data services
- Wireless and wireline base stations as well as the corresponding infrastructure
- Speech coding, synthesis, and voice recognition
- Wireless internet and multimedia
- Network and data communication



1.2 Architectural Differentiation

The SC140 architecture differentiates itself in the market with the following capabilities:

- **High-level Abstraction of the Application Software**
 - DSP applications and kernels can currently be developed in the C programming language. An optimizing compiler generates parallel instructions while maintaining a high code density.
 - An orthogonal instruction set and programming model along with single data space and byte addressability enable the compiler to generate efficient code.
 - Hardware supported integer and fractional data types enable application developers to choose their own style of code development, or to use coding techniques derived from an application-specific standard.
- **Scalable Performance**
 - The number of execution units is independent of the instruction set, and can be tailored to the application's performance requirement. The SC140 contains four arithmetic logic units (ALUs) and two address arithmetic units (AAUs).
 - A high frequency of operation is achieved at low voltage, providing four million multiply and accumulate (MAC) operations per second (4 MMACS) for each megahertz of clock frequency.
 - Support exists for application-specific accelerators, providing a performance boost and reduction in power consumption.
- **High Code Density for Minimized Cost**
 - 16-bit wide instruction encoding.
 - A rich and orthogonal instruction set, major portions of which focus on control code that can often occupy most of the application code.
 - Variable length execution set (VLES) for DSP kernel operations.
- **Improved Support for Multi-tasking Applications**
 - Dual stack pointer support in HW.
- **Optimized Power Management Control**
 - Very low power consumption.
 - Low voltage operation.
 - Power saving modes.
- **Efficient Memory and I/O Interface**
 - Very large on-chip zero-wait state static random access memory (SRAM) capability.
 - Support for slower on-chip memory via wait-states.
 - 32-bit address space for both program and data (byte-addressable).
 - Unified data and program memory space.
 - Decoupled external memory timing with independent clock.
- **Core Organization and Design**
 - Supports flexible system-on-a-chip (SoC) configurations.
 - Portable across fabrication lines and foundries.



1.3 Core Architecture Features

The SC140 core consists of the following:

- Data arithmetic logic unit (DALU) that contains four instances of an arithmetic logic unit (ALU) and a data register file
- Address generation unit (AGU) that contains two address arithmetic units (AAU) and an address register file
- Program sequencer and control unit (PSEQ)

Key features of the SC140 core include the following:

- Up to four million multiply-accumulate (MAC) operations per second (4 MMACS) for each megahertz of clock frequency
- Up to 10 RISC MIPS (million instruction words per second) for each megahertz of clock frequency (a MAC operation is counted as two RISC instructions)
- Four ALUs comprising MAC and bit-field units
- A true $(16 * 16) + 40$ to 40-bit MAC unit in each ALU
- A true 40-bit parallel barrel shifter in each ALU
- Sixteen 40-bit data registers for fractional and integer data operand storage
- Sixteen 32-bit address registers, eight of which can be used as 32-bit base address registers
- Four address offset registers and four modulo address registers
- Hardware support for fractional and integer data types
- Up to six instructions executed in a single clock cycle
- Very rich 16-bit wide orthogonal instruction set
- Support for application specific instruction set enhancements with an interface to an ISAP (Instruction Set Accelerator Plug-in)
- VLES execution model
- Two AAUs with integer arithmetic capabilities
- A bit mask unit (BMU) for bit and bit-field logic operations
- Unique DSP addressing modes
- 32-bit unified data and program address space
- Zero-overhead hardware loops with up to four levels of nesting
- Byte-addressable data memory
- Position independent code utilizing change-of-flow instructions that are relative to the program counter (PC)
- Enhanced on-chip emulation (EOnCE) module with real-time debug capabilities
- Low power wait standby mode
- Very low power complementary metal-oxide semiconductor (CMOS) design
- Fully static logic



1.3.1 Typical System-On-Chip Configuration

The SC140 is a high-performance general-purpose fixed-point DSP core, allowing it to support many system-on-chip (SoC) configurations. A library of modules containing memories, peripherals, accelerators, and other processor cores makes it possible for a variety of highly integrated and cost-effective SoC devices to be built around the SC140. Figure 1-1 shows a block diagram of a typical SoC chip made up of the SC140 core and associated SoC components (described below). In a typical system the SC140 core is enveloped in a platform that includes the core and supporting zero wait-state memories. This platform is integrated as a unit in the SoC. Although not indicated in this configuration, an SoC can contain more than one SC140 core platform.

An on-platform instruction set accelerator plug-in can be used as part of the SC140 core platform to provide additional instructions for unique application solutions such as video processing, which require specific arithmetic instructions in addition to the main instruction set.

- **SC140 DSP core platform** — Includes the DSP core and the immediate supporting blocks that typically run at the full core frequency. The DSP platform typically includes:
 - SC140 DSP core
 - Instruction Set Accelerator Plug-in (ISAP) - for expanding the instruction set with application-specific instructions.
 - L1 caches - data and instruction caches, operating with zero wait states in case of cache hit
 - Unified M1 memory - supporting both program and data, and hence connected to both the program and data buses of the core. The M1 memory operates with no wait states. It could be either RAM or ROM, or a mix of both. The RAM, depending on its' size, may be connected as a slave to an external DMA.
 - Program interrupt controller (PIC)
 - Interfaces - translate the core data and program fetch requests to the bus protocol supported by the system, usually in reduced frequency.
- **DSP Expansion Area** — This area includes the functional units that interface between the core and the DSP application, most importantly the functions that send and receive data from external input/output sources, under the control of the software running on the DSP core. In addition, this area includes accelerators that execute portions of the application, in order to boost performance and decrease power consumption. This area is application-specific and may or may not include various functional units such as:
 - Synchronous serial interface
 - Serial communication interface
 - Viterbi accelerator
 - Filter coprocessors
- **System Expansion Area** — This area includes the SoC functional units that are not tightly coupled with the DSP core. Typically it may include other processors with their support platform as well. This area is application-specific, and may include various functional units such as:
 - External memory interface
 - Direct memory access (DMA) controller
 - L2 Cache controller for either data or program
 - Chip-level Interrupt control unit
 - On-chip Level 2 (M2) memory expansion modules
 - Other processor cores with their supporting platforms

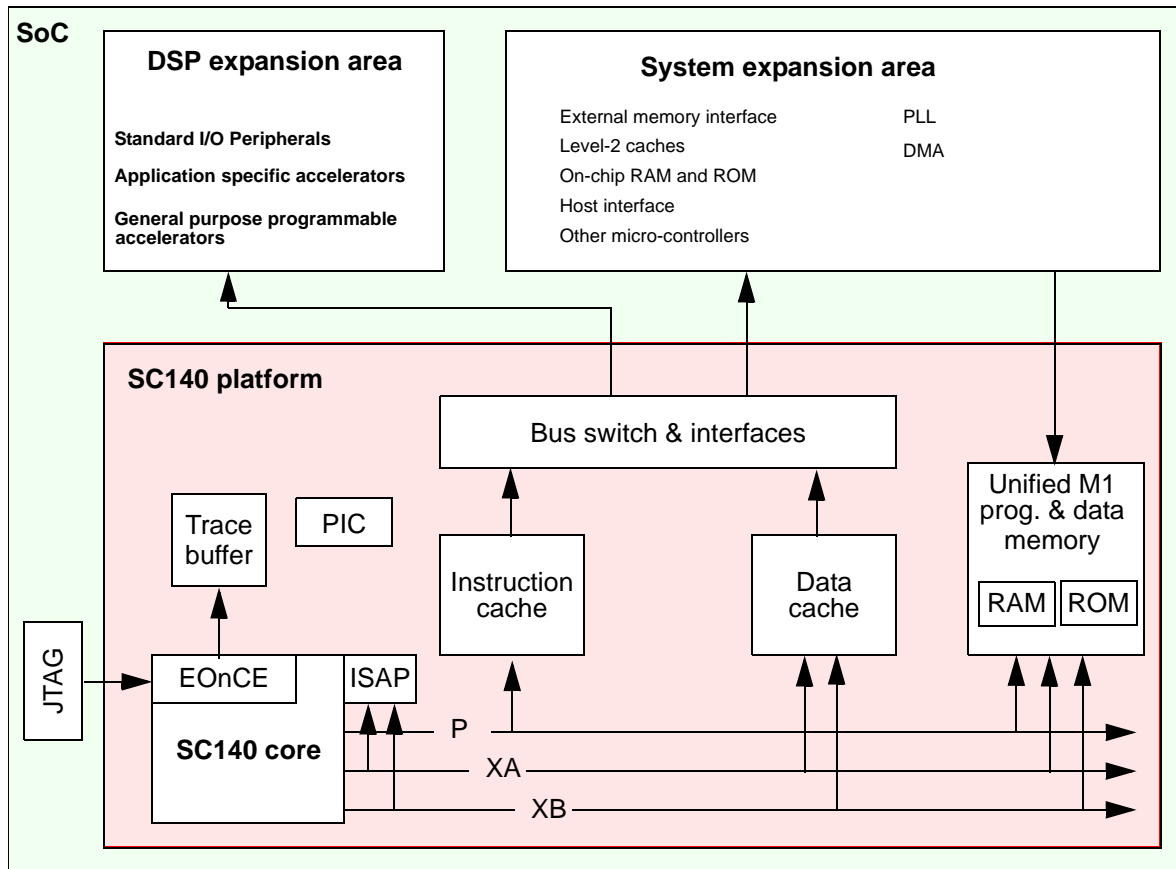


Figure 1-1. Block Diagram of a Typical SoC Configuration with the SC140 Core

1.3.2 Variable Length Execution Set (VLES) Software Model

The VLES software model is the instruction grouping used by the SC140 to address the requirements of DSP kernels. Using an orthogonal compiler-friendly instruction set, this model maintains a compact code density for applications.

All SC140 instruction words are 16 bits wide. Most instructions are encoded with one word. Each SC140 instruction encodes an atomic (lowest-level) operation. For example, MAC and store (move) instructions are encoded in 16 bits. Since atomic operations need fewer bits to encode, the 16-bit instruction set becomes fully orthogonal and very rich in the functionality it supports.

In order to execute signal processing kernels, a set of SC140 instructions can be grouped to be executed in parallel. The PSEQ performs this automatically with up to four DALU instructions and two AGU instructions executed at the same time.



Chapter 2

Core Architecture

This chapter provides an overview of the SC140 core architecture. It describes the main functional blocks and data paths of the core.

2.1 Architecture Overview

The SC140 core provides the following main functional units:

- Data arithmetic and logic unit (DALU)
- Address generation unit (AGU)
- Program sequencer unit (PSEQ)

To provide data exchange between the core and the other on-chip blocks, the following buses are implemented:

- Two data memory buses (address and data pairs: XABA and XDBA, XABB and XDBB) that are used for all data transfers between the core and memory.
- Program data and address buses (PDB and PAB) for carrying program words from the memory to the core.
- Special buses to support tightly coupled external user-definable instruction set accelerators.

A block diagram of the SC140 core is shown in Figure 2-3.

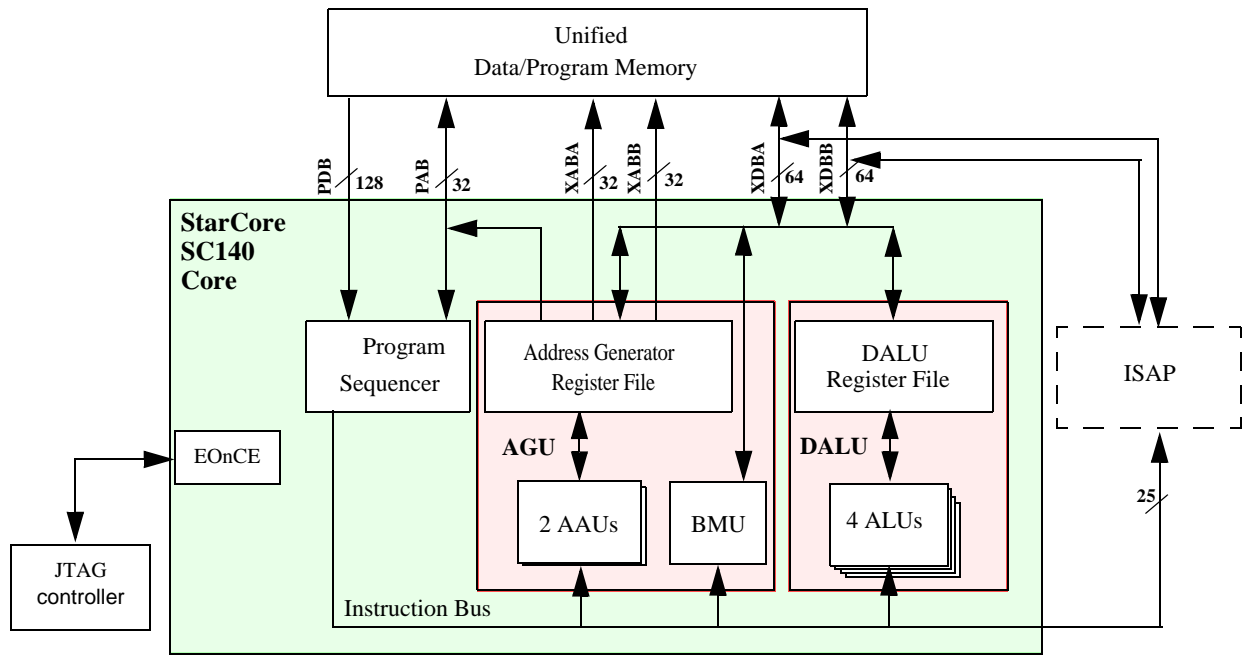


Figure 2-1. Block Diagram of the SC140 Core

2.1.1 Data Arithmetic Logic Unit (DALU)

The DALU performs arithmetic and logical operations on data operands in the SC140 core. The components of the DALU are as follows:

- A register file of sixteen 40-bit registers
- Four parallel ALUs, each ALU containing a multiply-accumulate (MAC) unit and a bit-field unit (BFU)
- Eight data bus shifter/limiters

All the MAC units and BFUs can access all the DALU registers. Each register is partitioned into three portions: two 16-bit registers (low and high portion of the register) and one 8-bit register (extension portion). Accesses to or from these registers can be in widths of 8 bits, 16 bits, 32 bits, or 40 bits, depending on the instruction.

The two data buses between the DALU register file and the memory are each 64 bits wide. This enables a very high data transfer speed between memory and registers by allowing two data moves in parallel, each up to 64 bits in width. The move instructions vary in access width from 8 bits to 64 bits, and can transfer multiple words within the 64 bit constraint. With every MOVE instruction affecting the memory, one of four signals to the memory interface is asserted, defining the access width.

- MOVE.B loads or stores bytes (8-bit).
- MOVE.W or MOVE.F loads or stores integer or fractional words (16-bit).
- MOVE.2W, MOVE.2F or MOVE.L loads or stores two integers, two fractions and long words respectively (32-bit).
- MOVE.4W or MOVE.4F loads or stores four integers or four fractions, respectively (64-bit).

- MOVE.2L loads or stores two long words (64-bit).

2.1.1.1 Data Register File

The DALU registers can be read or written over the data buses (XDBA and XDBB). A DALU register can be the source for up to four simultaneous instructions, but simultaneous writes of a destination register are illegal. The source operands for DALU arithmetic instructions usually originate from DALU registers. The destination of every arithmetic operation is a DALU register, and each such destination can be used as a source operand for the operation immediately following, without any time penalty.

2.1.1.2 Multiply-Accumulate (MAC) Unit

The MAC unit comprises the main arithmetic processing unit of the SC140 core and performs the arithmetic operations. The MAC unit has a 40-bit input and outputs one 40-bit result in the form of [Extension:High Portion:Low Portion] (EXT:HP:LP).

The multiplier executes 16-bit by 16-bit fractional or integer multiplication between two's complement signed, unsigned, or mixed operands (16-bit multiplier and multiplicand). The 32-bit product is right-justified, sign-extended, and may be added to the 40-bit contents of one of the 16 data registers.

2.1.1.3 Bit-Field Unit (BFU)

The BFU contains a 40-bit parallel bidirectional shifter with a 40-bit input and a 40-bit output, a mask generation unit, and a logic unit. The BFU is used in the following operations:

- Multi-bit left/right shift (arithmetic or logical)
- One-bit rotate (right or left)
- Bit-field insert and extract
- Count leading bits (ones or zeros)
- Logical operations
- Sign or zero extension operations

2.1.1.4 Shifter/Limiters

Eight shifter/limiters provide scaling and limiting on 32-bit transfers from the data register file to memory. Scaling up or down by one bit is programmable as is limiting to the maximum values provided in 32 bits. For more detailed information, see [Section 2.2.1.4, “Data Shifter/Limiter,”](#) [Section 2.2.1.5, “Scaling,”](#) and [Section 2.2.1.6, “Limiting.”](#)

2.1.2 Address Generation Unit (AGU)

The AGU contains address registers and performs address calculations using integer arithmetic necessary to address data operands in memory. It implements four types of arithmetic: linear, modulo, multiple wrap-around modulo, and reverse-carry. The AGU operates in parallel with other core resources to minimize address generation overhead.

The AGU in the SC140 core has two address arithmetic units (AAU) to allow two address generation operations at every clock cycle. The AAU has access to:

- Sixteen 32-bit address registers (R0–R15), of which R8–R15 can also be used as base address registers for modulo addressing.
- Four 32-bit offset registers (N0–N3).
- Four 32-bit modulo registers (M0–M3).

The two AAUs are identical. Each contains:

- A 32-bit full adder, used for offset calculations.
- A second 32-bit full adder, used for modulo calculations.

Each AAU can update one address register in the address register file in one instruction cycle.

The AGU also contains a 32-bit modulo control register (MCTL). This control register is used to specify the addressing mode of the R registers: linear, reverse-carry, modulo, or multiple wrap-around modulo. When modulo addressing mode is selected, the MCTL register is used to specify which of the four modulo registers is assigned to a specific R register.

Explicit instructions in the SC140 instruction set are used to execute arithmetic operations on the address pointers. This capability can also be used for general data arithmetic. In addition, the AGU generates change-of-flow program addresses and updates the stack pointers as needed.

2.1.2.1 Stack Pointer Registers

Two special registers with special addressing modes are used for software stacks. These are the Normal mode stack pointer (NSP) and the Exception mode stack pointer (ESP). Both the ESP and the NSP are 32-bit read/write address registers with pre-decrement and post-increment updates. Both are offset with immediate values to allow random access to a software stack.

The ESP is used by stack instructions when the SC140 is in the Exception mode of operation, which is entered when exceptions occur. The NSP is used in Normal mode when there are no exceptions. The existence of two stack pointers enables separate allocation of stack space by the operating system and each application task, which optimizes memory use in multi-tasking systems.

2.1.2.2 Bit Mask Unit (BMU)

The BMU provides an easy way of setting, clearing, inverting, or testing a selected, but not necessarily adjacent, group of bits in a register or memory location.

The BMU supports a set of bit mask instructions that operate on:

- All AGU pointers (R0–R15)
- All DALU registers (D0–D15)
- All control registers (EMR, VBA, SR, MCTL)
- Memory locations

Only a single bit mask instruction is allowed in any single execution set since only one execution unit exists for these instructions.

A subgroup of the bit mask instructions (BMTSET) provides hardware support of semaphoring, providing one instruction for read-modify-write.



2.1.3 Program Sequencer Unit (PSEQ)

The PSEQ performs instruction fetch, instruction dispatch, hardware loop control, and exception processing. The PSEQ controls the different processing states of the SC140 core. The PSEQ consists of three hardware blocks:

- Program dispatch unit (PDU)—Responsible for detecting the execution set out of a one or two fetch set, and dispatching the execution set's various instructions to their appropriate execution units where they are decoded.
- Program control unit (PCU)—Responsible for controlling the sequence of the program flow.
- Program address generator (PAG)—Responsible for generating the program counter (PC) for instruction fetch operations, including hardware looping.

The PSEQ implements its functions using the following registers:

- PC—Program counter register
- SR—Status register
- SA0-3—Four start address registers (SA0–SA3)
- LC0-3—Four loop counter registers (LC0–LC3)
- EMR—Exception and mode register
- VBA—Interrupt vector base address register

2.1.4 Enhanced On-Chip Emulator (EOnCE)

The EOnCE module provides a non-intrusive means of interacting with the SC140 core and its peripherals so that a user can examine registers, memory, or on-chip peripherals as well as define various breakpoints and read the trace-FIFO. The EOnCE module greatly aids the development of hardware and software on the SC140 core processor, EOnCE interfacing with the debugging system through on-chip JTAG TAP controller pins. Refer to [Chapter 4, “Emulation and Debug \(EOnCE\),”](#) for details.

2.1.5 Instruction Set Accelerator Plug-in (ISAP) Interface

A user-defined instruction set accelerator plug-in (ISAP) module provides a means of enhancing the SC140 basic instruction set with additional instructions. These additional instructions are executed in an external module connected to the core. The new instructions are added to the SC140 Assembler and Compiler via intrinsic libraries making application-specific or general-purpose functions available to the user. A 25-bit instruction bus from the SC140 core to the ISAP enables the definition and support of a very rich instruction set. The ISAP is also connected to the two 64-bit data buses, providing a large data bandwidth to the main memory system.

2.1.6 Memory Interface

The SC140 core uses a unified memory space. Each address can contain either program information or data. The exact memory configuration is customizable for each chip containing an SC140 core. Memory space typically consists of on-chip RAM and ROM that can be expanded off-chip. The memory system must support two parallel data accesses. However, it may issue stalls due to its specific implementation. Refer to [Section 2.4, “Memory Interface,”](#) for further details.

Both internal and external memory configurations are specific to each member of the SC140 family.

2.2 DALU

This section describes the architecture and operation of the DALU, the block where most of the arithmetic and logical operations are performed on data operands. In addition, this section details the arithmetic and rounding operations performed by the DALU as well as its programming model.

2.2.1 DALU Architecture

The DALU performs most of the arithmetic and logical operations on data operands in the SC140 core.

The data registers can be read from or written to memory over the XDBA and the XDBB as 8-bit, 16-bit, or 32-bit operands. The 64-bit wide data buses, XDBA and XDBB, support the transfer of several operands in a single access. The source operands for the DALU, which may be 16, 32, or 40 bits, originate either from data registers or from immediate data. The results of all DALU operations are stored in the data registers.

All DALU operations are performed in one clock cycle. Up to parallel arithmetic operations can be performed in each cycle. The destination of every arithmetic operation can be used as a source operand for the operation immediately following without any time penalty.

The components of the DALU are as follows:

- A register file of sixteen 40-bit registers
- Four parallel ALUs, each containing a MAC unit and a BFU with a 40-bit barrel shifter
- Eight data bus shifter/limiters that allow scaling and limiting of up to four 32-bit operands transferred over each of the XDBA and XDBB buses in a single cycle

Figure 2-2 shows the architecture of the DALU.

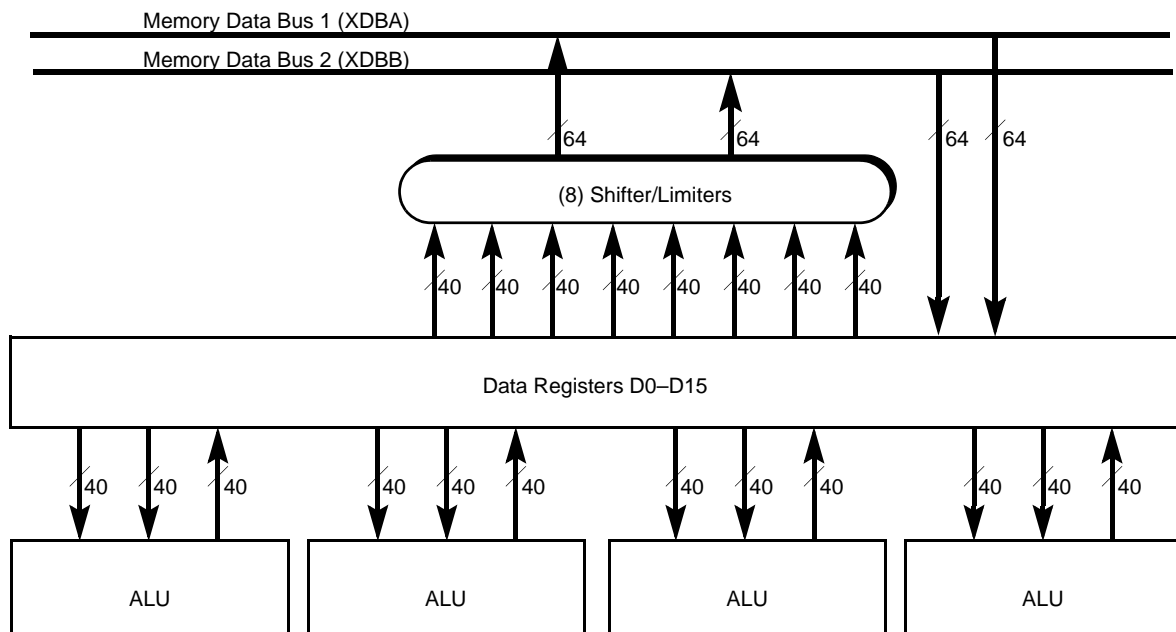


Figure 2-2. DALU Architecture



The DALU programming model is shown in Table 2-1. Register D0 refers to the entire 40-bit register, whereas D0.e, D0.h, and D0.l refer to the extension: high portion and low portion of the D0 register, respectively. In addition, one limit tag bit is associated with each data register. L0–L15 are concatenated to D0–D15, respectively.

Table 2-1. DALU Programming Model

LIMIT	EXT	HP	LP
L0	D0.e	D0.h	D0.l
L1	D1.e	D1.h	D1.l
L2	D2.e	D2.h	D2.l
L3	D3.e	D3.h	D3.l
L5	D5.e	D5.h	D5.l
L6	D6.e	D6.h	D6.l
L7	D7.e	D7.h	D7.l
L8	D8.e	D8.h	D8.l
L9	D9.e	D9.h	D9.l
L10	D10.e	D10.h	D10.l
L11	D11.e	D11.h	D11.l
L12	D12.e	D12.h	D12.l
L13	D13.e	D13.h	D13.l
L14	D14.e	D14.h	D14.l
L15	D15.e	D15.h	D15.l

2.2.1.1 Data Registers (D0–D15)

In this section, the D0–D15 data registers are referred to as Dn. They can be used as:

- Source operands
- Destination operands
- Accumulators

The registers can serve as input buffer registers between XDBA or XDBB and the ALUs. The registers are used as DALU source operands, allowing new operands to be loaded for the next instruction while the register contents are used by the current arithmetic instruction.

Each data register Dn has a limit tag bit (Ln) which is used to signify whether the extension portion of the register is in use. The limit tag bit Ln is coupled to the extension portion Dn.e, which forms a 9-bit operand for the purpose of storing these bits to memory. See [Section 2.2.1.6, “Limiting,”](#) for further details.

The data registers can be accessed over XDBA and XDBB with three data widths:

- A long-word access, writing or reading 32-bit operands
- A word access, writing or reading 16-bit operands
- A byte access, writing or reading 8-bit operands

For move instructions of fractional data, the transfer of a Dn register to memory over XDBA and XDBB is protected against overflow by substituting a limiting constant for the data that is being transferred. The content of Dn is not affected should limiting occur. Only the value transferred over XDBA or XDBB is limited. This process is commonly referred to as transfer saturation and should not be confused with the arithmetic saturation mode as described in [Section 2.2.2.7, “Arithmetic Saturation Mode.”](#)

Limiting is performed after the contents of the register have been shifted according to the scaling mode. Shifting and limiting are performed only for MOVES instructions when a fractional operand is specified as the source for a data move over XDBA or XDBB. When an integer operand is specified as the source for a data move, shifting and limiting are not performed.

Automatic sign extension (or zero extension of the data values into the 40-bit registers) is provided when an operand is transferred from memory to a data register. Sign extension can occur when loading the Dn register from memory. If a fractional word operand is to be written to a data register, the high portion (HP) of the register is written with the word operand. The low portion (LP) is zero-filled. The EXT portion is sign-extended from the HP, and the limit tag bit (Ln) is cleared.

When an integer word operand is to be written to a data register, the LP portion of the register is written with the word operand. The HP and EXT portions are either zero-extended or sign-extended from the LP. Long-word operands are written into the HP:LP portions of the register. The EXT portion is zero-extended or sign-extended, and the limit tag bit (Ln) is cleared.

When a byte operand is to be written to a data register, the register’s first 8-bit portion of the LP (Dn.1[7:0]) is written with the byte operand. The following eight bits of the LP (Dn.1[15:8]), the high portion, and the EXT are either zero-extended or sign-extended from the LP lower byte. The limit tag bit (Ln) is cleared.



A special case of the MOVE.L instruction is used for reading from or writing to the EXT portion of a data register. Six variations of this instruction save (restore) the extension bits and Ln bit of data registers to (from) memory. One of the variations writes to memory the Ln bit and extension bits of an even and an odd pair of registers. Another variation reads bits 8:0 from memory to the extension bits and the Ln bit of an even register. Another variation reads bits 24:16 to the extension bits and the Ln bit of an odd register. Memory writes are done from the even/odd pair of registers. Memory reads are done to a single register. An extension saved to memory from an even numbered register must be restored to an even register, likewise for odd registers.

All move instructions are described in detail in [Appendix A, “SC140 DSP Core Instruction Set.”](#)

Table 2-2 summarizes the various types of data bus write access to the data registers.

Note: When an unsigned long operand is written to a data register, Dn.e is zero-extended.

Table 2-2. Write to Data Registers

Operand Type	Ln	Dn.e	Dn.h	Dn.l
Fractional word	Zero-extended	Sign-extended	Operand	Zero-filled
Integer Byte	Zero-extended	Zero-extended/ Sign-extended	Zero-filled/ Sign-extended	Upper byte - Sign-extended/zero-extended Lower byte - Operand
Integer Word	Zero-extended	Zero-extended/ Sign-extended	Zero-filled/ Sign-extended	Operand
Long	Zero-extended	Zero-extended/ Sign-extended	Operand	Operand
2 Extensions - Long	Operand	Operand	Unchanged	Unchanged

Table 2-3 summarizes the various types of data bus read accesses from the data registers.

Table 2-3. Read from Data Registers

Operand Type	Memory Data Bus.h	Memory Data Bus.l	Limiting/Scaling
Fractional Word	-	Dn.h	Yes/No (See Note)
Fractional Long	Dn.h	Dn.l	Yes/No (See Note)
Integer Word	-	Dn.l	No
Integer Long	Dn.h	Dn.l	No
Integer Byte	-	Low byte - Dn.l[7:0]	No
2 Extensions - Long	EXT word: {7 zero bits, L _{n+1} , D _{n+1.e} }	EXT word: {7 zero bits, Ln, Dn.e}	No

Note: A fractional word or fractional long word can be written to memory with or without limiting and shifting. See MOVE.F and MOVES.F in [Appendix A, “SC140 DSP Core Instruction Set.”](#)

The register file architecture and the 64-bit wide data buses XDBA and XDDB support wide data transfers between the memory and the data registers. Up to four 16-bit words or two 32-bit long words can be transferred between the register file and the memory in a single move operation on each data bus, XDBA or XDDB.

Table 2-4 summarizes the various data widths for data moves from/to the data register file.

Table 2-4. Data Registers Access Width

Operand Type	Data Width (Bits)
Byte	8
Word	16
Long	32
Two word	32
Four byte	32
Two long word	64
Four word	64

2.2.1.2 Multiply-Accumulate (MAC) Unit

The MAC unit is the arithmetic part of the ALU containing both a multiplier and an adder. It also performs other operations such as rounding, saturation, comparisons, and shifting. Inputs to the MAC unit are from data registers or from immediate data programmed into the instruction. As many as three operands may be inputs. The destination for MAC instructions is always a data register in the 40-bit form EXT:HP:LP. The multiplier executes 16 by 16 parallel multiplication of two's complement data, signed or unsigned, fractional or integer. The multiplier output can be accumulated with 40-bit data in a destination register. A detailed description of each multiplication operation is given in [Section 2.2.2.3, "Multiplication."](#) The adder executes addition and subtraction of two 40-bit operands. All MAC instructions are executed in one clock cycle.

Table 2-5 lists the arithmetic instructions that are executed in the MAC unit. A more detailed description of each instruction is given in [Appendix A, "SC140 DSP Core Instruction Set."](#)

Table 2-5. DALU Arithmetic Instructions (MAC)

Instruction	Description
ABS	Absolute value
ADC	Add long with carry
ADD	Add
ADD2	Add two words
ADDNC.W	Add without changing the carry bit in the SR
ADR	Add and round
ASL	Arithmetic shift left by one bit
ASR	Arithmetic shift right by one bit
CLR	Clear
CMPEQ	Compare for equal
CMPGT	Compare for greater than
CMPHI	Compare for higher (unsigned)

Table 2-5. DALU Arithmetic Instructions (MAC) (Continued)

Instruction	Description
DECEQ	Decrement a data register and set T (the true bit) if zero
DECGE	Decrement a data register and set T if greater than or equal to zero
DIV	Divide iteration
DMACSS	Multiply signed by signed and accumulate with data register right-shifted by word size
DMACSU	Multiply signed by unsigned and accumulate with data register right-shifted by word size
IADDNC.W	40-bit non-saturating add integers with immediate, no carry update
IMAC	Multiply-accumulate integers
IMACLHUU	Multiply-accumulate unsigned integers: first source from low portion, second from high portion
IMACUS	Multiply-accumulate unsigned integer and signed integer
IMPY.W	Multiply integer
IMPYHLUU	Multiply unsigned integer and unsigned integer: first source from high portion, second from low portion
IMPYSU	Multiply signed integer and unsigned integer
IMPYUU	Multiply unsigned integer and unsigned integer
INC	Increment a data register
INC.F	Increment a data register (as fractional data)
MAC	Multiply-accumulate signed fractions
MACR	Multiply-accumulate signed fractions and round
MACSU	Multiply-accumulate signed fraction and unsigned fraction
MACUS	Multiply-accumulate unsigned fraction and signed fraction
MACUU	Multiply-accumulate unsigned fraction and unsigned fraction
MAX	Transfer maximum signed value
MAX2	Transfer two 16-bit maximum signed values
MAX2VIT	Transfer two 16-bit maximum signed values, update Viterbi flags
MAXM	Transfer maximum magnitude value
MIN	Transfer minimum signed value
MPY	Multiply signed fractions
MPYR	Multiply signed fractions and round
MPYSU	Multiply signed fraction and unsigned fraction
MPYUS	Multiply unsigned fraction and signed fraction
MPYUU	Multiply unsigned fraction and unsigned fraction

Table 2-5. DALU Arithmetic Instructions (MAC) (Continued)

Instruction	Description
NEG	Negate
RND	Round
SAT.F	Saturate fractional value in data register to fit in high portion
SAT.L	Saturate value in data register to fit in 32 bits
SBC	Subtract long with carry
SBR	Subtract and round
SUB	Subtract
SUB2	Subtract two words
SUBL	Shift left and subtract
SUBNC.W	Subtract with no carry bit generation
TFR	Transfer data register to a data register
TFRF	Transfer data register to a data register if T bit is false
TFRT	Transfer data register to a data register if T bit is true
TSTEQ	Test for equal to zero
TSTEQ.L	32-bit compare for equal to zero
TSTGE	Test for greater than or equal to zero
TSTGT	Test for greater than zero

2.2.1.3 Bit-Field Unit (BFU)

The BFU is the logic part of the ALU. It contains a 40-bit parallel bidirectional shifter (with a 40-bit input and a 40-bit output) mask generation unit and logic unit. The BFU is used in the following operations:

- Multi-bit left/right shift (arithmetic or logical)
- One-bit rotate (right or left)
- Bit-field insert and extract
- Count leading bits (ones or zeros)
- Logical operations
- Sign or zero extension operations

Table 2-6 lists the instructions which are executed in the BFU. A more detailed description of each instruction is given in [Appendix A, “SC140 DSP Core Instruction Set.”](#)

Table 2-6. DALU Logical Instructions (BFU)

Instruction	Description
AND	Logical AND
ASLL	Multi-bit arithmetic shift left
ASLW	Word arithmetic shift left (16-bit shift)
ASRR	Multi-bit arithmetic shift right
ASRW	Word arithmetic shift right (16-bit shift)
CLB	Count leading bits (ones or zeros)
EOR	Bit-wise exclusive OR
EXTRACT	Extract signed bit-field
EXTRACTU	Extract unsigned bit-field
INSERT	Insert bit-field
LSLL	Multi-bit logical shift left
LSR	Logical shift right by one bit
LSRR	Multi-bit logical shift right
LSRW	Word logical shift right (16-bit shift)
NOT	One's complement (inversion)
OR	Bit-wise inclusive OR
ROL	Rotate one bit left through the carry bit
ROR	Rotate one bit right through the carry bit
SXT.B	Sign extend byte
SXT.L	Sign extend long
SXT.W	Sign extend word
ZXT.B	Zero extend byte
ZXT.L	Zero extend long
ZXT.W	Zero extend word

2.2.1.4 Data Shifter/Limiter

The data shifters/limiters provide special post-processing on data written from a Dn register to the XDBA or XDBB buses. There are eight independent shifters/limiters, four for the XDBA bus and four for the XDBB bus, allowing transfers to memory of up to four words per MOVES instruction with scaling and limiting. Each consists of a shifter for scaling followed by a limiter. Note that arithmetic saturation from DALU operations is a different function. Saturation occurs in the DALU before data is written to a destination register.

2.2.1.5 Scaling

The data shifters in the shifter/limiter unit can perform the following data shift operations:

- Scale up—Shift data one bit to the left
- Scale down—Shift data one bit to the right
- No scaling—Pass the data unshifted

The eight shifters permit direct dynamic scaling of fixed-point data without additional program steps. For example, this permits straightforward block floating-point implementation of Fast Fourier Transforms (FFTs).

Scaling occurs if programmed in the scaling mode bits S0 and S1 (bits 4 and 5 in the SR). Scaling of operands only occurs with the MOVES.F, MOVES.2F, MOVES.4F, and MOVES.L instructions, moving data from a DALU register (or registers) to memory. The data in the register is not changed, only the data that is transferred. The scaling mode also affects the Ln bit calculation and the rounding function for a set of DALU instructions. Scaling is disabled when the arithmetic saturation mode is set. See [Section 3.1.1, “Status Register \(SR\),”](#) and below for further details. An example of scaling is provided in Table 2-7.

Table 2-7. Scaling Example

Instruction	Memory/ Register	New Value	Comments
move.w #\$0030,r0	r0	\$0000 0030	R0 initialized for first memory write
moveu.w #\$0200,d0.h	d0	\$0200 0000	D0 written
bmset #\$10,sr.l	sr	\$0000 0010	Scale down set in SR
moves.f d0,(r0)+	\$0030	\$0100	Memory written with scaled down value
move.l #\$00e40020,sr	sr	\$00e4 0020	Scale up set in SR
moves.f d0,(r0)	\$0032	\$0400	Memory written with scaled up value

2.2.1.6 Limiting

The limiting capability is enabled only for the MOVES.F, MOVES.2F, MOVES.4F, and MOVES.L instructions, and not for any other fractional moves such as MOVE.F. These instructions move data from DALU register(s) to memory. The limiting operation takes place in two steps: first, calculating the Ln bit when a previous ALU instruction wrote to a register, and second, transferring the data from that register with a MOVES instruction. The transferred data is limited if the Ln bit is set.

2.2.1.6.1 Calculating the Ln Bit

The Ln bit can be affected by ALU instructions which are capable of using the extension portion of a data register. The only use of the Ln bit is to set up or prepare for a subsequent MOVES instruction. The Ln bit is calculated based on the effective extension bits shown in Table 2-8. These are the bits to the left of the implied decimal point after scaling. If the bits are not all zeros or all ones, the extension is effectively in use and the Ln bit will be set. The Ln bit is cleared as data is written to a DALU register if the defining bits below are all zeros or all ones.

Table 2-8. Ln Bit Calculation

S1	S0	Scaling Mode	Bits Defining the Ln bit Calculation
0	0	No Scaling	Bits 39, 38.....32, 31
0	1	Scale Down	Bits 39, 38.....33, 32
1	0	Scale Up	Bits 39, 38.....31, 30

The Ln bit is calculated (and set or cleared) for the following saturable instructions: ABS, ADC, ADR, ADD, ADDNC, ASL, ASR, DIV, INC, MAC, MACR, MPY, MPYR, NEG, RND, SBC, SBR, SUB, SUBL, SUBNC, and TFRx. The Ln bit is cleared if arithmetic saturation mode is set, except for these instructions: ADC, DIV, SBC, TFR, TFRT, and TFTF. For the latter six, the Ln bit calculation is done, even if arithmetic saturation mode is set. However, no scaling is considered in the Ln bit calculation if the arithmetic saturation mode is set, even if a scaling mode bit is set.

The Ln bit is always cleared as a result of the execution of one of the following instructions: CLR, DECEQ, DECGE, MAX, MAXM, MIN, ADD2, SUB2, MAX2, MAX2VIT, DMAC_{su}, DMAC_{ss}, MAC_{su}, MAC_{uu}, MAC_{us}, MPY_{su}, MPY_{uu}, MPY_{us}, IADDNC, SAT, all integer multiplication operations, all BFU operations (as listed in Table 2-6 on page 2-13), and all MOVE instructions except for the specialized MOVE instruction that restores (pops the stack) the extension and Ln bits from memory. If the result of these instructions is required to be limited by a following move operation (a TFR Dn), the Dn instruction should be executed after the original instruction in order to validate the Ln bit before the value is written to memory using a MOVES.x operation.

2.2.1.6.2 Limiting with the MOVES Instructions

The second stage of limiting occurs with the execution of a MOVES instruction. A limited value is substituted for the transferred data if the Ln bit of that register was set. The data in the register is not changed, only the data transferred.

Having four limiters for each bus allows eight operands to be limited independently in the same instruction cycle. The four data limiters per bus can also be combined to form two 32-bit data limiters per bus for long-word operands.

If limiting occurs, the data limiter substitutes a limited data value having maximum magnitude (saturated) and the same sign as the 40-bit source register content:

- \$7FFF for 16-bit positive numbers
- \$7FFF FFFF for 32-bit positive numbers
- \$8000 for 16-bit negative numbers
- \$8000 0000 for 32-bit negative numbers

This substitution process is sometimes called transfer saturation. The value in the register is not shifted or limited, and can be reused by subsequent instructions. If the arithmetic saturation mode is set in the SR, scaling is not considered in the calculation of the Ln bit. An example of limiting is provided in Table 2-9.

Table 2-9. Limiting Example

Instruction	Memory/ Register	New Value	Comments
move.w #\$0030,r0	r0	\$0000 0020	R0 holds the address for the first move to memory
moveu.w #\$7ff,d0.h	d0	\$7ff 0000	d0.h set with the most positive 2's complement number
moveu.w #\$7ff,d1.h	d1	\$7ff 0000	d1.h set with the most positive 2's complement number
add d0,d1,d3	d3	\$1:00:ffe 0000	L3 bit set from overflow
move.f d3,(r0)+	\$0020	\$ffe	No limiting from the move instruction
moves.f d3,(r0)	\$0022	\$7ff	Limiting occurs with the moves instruction

Note that in the unusual case where arithmetic saturation mode is set between a DALU instruction and a subsequent moves instruction, scaling with the moves instruction is inhibited. However, limiting will occur if the Ln bit is already set.

2.2.1.7 Scaling and Arithmetic Saturation Mode Interactions

The following table shows the scaling and limiting operations for the four possible cases of scaling/no scaling with arithmetic saturation mode on/off. Note that the mode of both scaling and arithmetic saturation selected is not a normal mode of operation for the core. The “Special Six” instructions referred to in Table 2-10 and Table 2-11 are ADC, DIV, SBC, TFR, TFRT, and TFTF.

Table 2-10. Scaling and Limiting Interactions

Scaling Selected	Arithmetic Saturation Mode	Ln Bit Calculation			Limiting with MOVES instructions (see note below)	Scaling with MOVES Instructions
		Saturable DALU Instructions	Special Six Instructions	Other DALU Instructions		
None	Off	Calculated, no scaling	Calculated, no scaling	Cleared	Yes	No
Up/down	Off	Calculated, with scaling	Calculated, with scaling	Cleared	Yes	Yes
Off	On	Cleared	Calculated, no scaling	Cleared	Yes	No
Up/down	On	Cleared	Calculated, no scaling	Cleared	Yes	No

Note: Limiting will occur if the Ln bit is set.



The following table (Table 2-11) shows the arithmetic saturation and rounding operations for the four possible cases of scaling, no scaling, and arithmetic saturation mode on/off.

Table 2-11. Saturation and Rounding Interactions

Scaling Selected	Arithmetic Saturation Mode	Arithmetic Saturation		Rounding
		Saturable DALU Instructions	Special Six Instructions	
None	Off	None	None	Rounding with no scaling
Up/down	Off	None	None	Rounding with scaling considered
None	On	Saturation can occur	None	Rounding with no scaling
Up/down	On	Saturation can occur, no scaling considered	None	Rounding with no scaling

2.2.2 DALU Arithmetic and Rounding

The following paragraphs describe the DALU data representation, rounding modes, and arithmetic methods.

2.2.2.1 Data Representation

The SC140 core uses either a fractional or integer two's complement data representation for all DALU operations. The main difference between fractional and integer representations is the location of the decimal (or binary) point. For fractional arithmetic, the decimal (or binary) point is always located immediately to the right of the most significant bit of the high portion. For integer values, it is always located immediately to the right of the least significant bit (LSB) of the value. Figure 2-3 shows the location of the decimal point (binary point) bit weighting and operand alignment for different fractional and integer representations supported on the SC140 architecture.

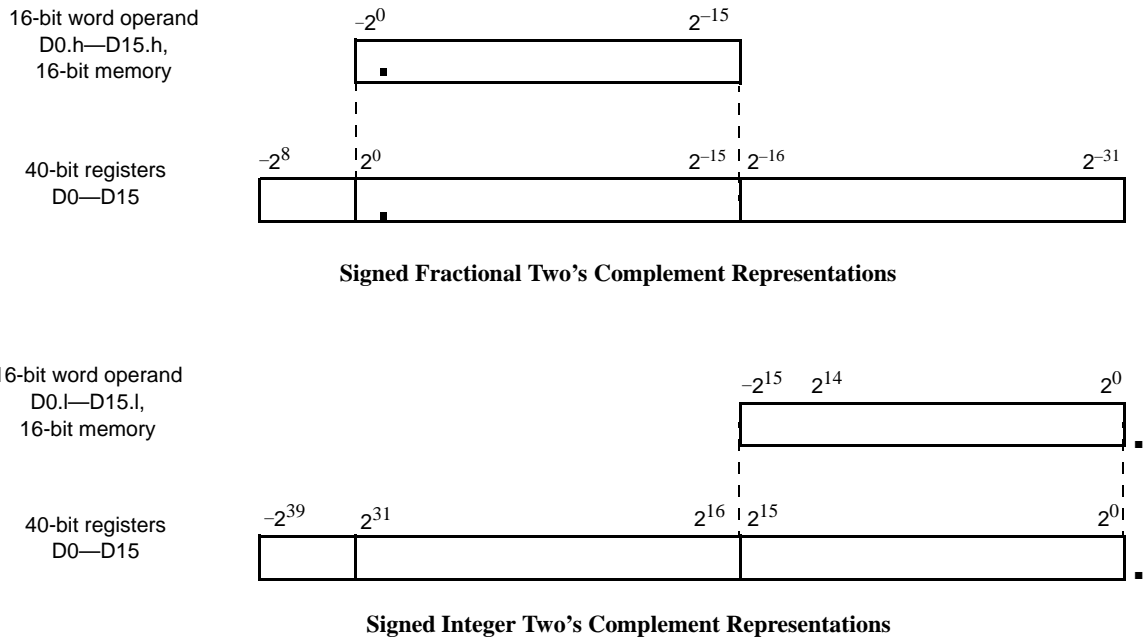


Figure 2-3. DALU Data Representations

2.2.2.2 Data Formats

Three types of two's complement data formats are supported by the SC140 core:

- Signed fractional (SF)
- Signed integer (SI)
- Unsigned integer (UI)

The ranges for each of these formats, described below, apply to all data stored in memory as well as data stored in the data registers. The extension associated with each register allows word growth so that the most positive fractional number that can be represented in a register is almost 256.0 with the most negative fractional number being exactly -256.0. When the register extension is in use, the data contained in the register cannot be stored exactly in memory or in other registers in a single move. In these cases, the storage error can be minimized by limiting the data to the most positive or most negative number consistent with the size of the destination, the sign of the register and the MSB of the extension.

2.2.2.2.1 Signed Fractional

In this format, without extension bits 39-32, the N-bit operand is represented using the 1.[N-1] bit format (1 sign bit, N-1 fractional bits). Signed fractional numbers lie in the following range:

$$-1.0 \leq SF \leq +1.0 - 2^{-[N-1]}$$

For words and long-word signed fractions, the most negative number that can be represented is exactly -1.0, of which the internal representation is \$8000 and \$8000 0000, respectively. The most positive word is \$7FFF or $1.0 - 2^{-15}$, and the most positive long word is \$7FFF FFFF or $1.0 - 2^{-31}$.

If the extension bits are in use, the most positive number is $256 - 2^{-31}$ represented by \$7F FFFF FFFF, and the most negative number is -256, represented by \$80 0000 0000.



2.2.2.2.2 Signed Integer

This format is used when processing data as integers. Using this format, the N-bit operand is represented using the N.0 bit format (N integer bits). Signed integer numbers lie in the following range:

$$-2^{[N-1]} \leq SI \leq [2^{[N-1]}-1]$$

For words and long-word signed integers, the most negative word that can be represented is -32768 (\$8000) and the most negative long word is -2147483648 (\$8000 0000). The most positive word is 32767 (\$7FFF) and the most positive long word is 2147483647 (\$7FFF FFFF).

If the extension bits are in use, N becomes 40, and the most positive number is $2^{39} - 1$ represented by \$7F FFFF FFFF. The most negative number is -2^{39} , represented by \$80 0000 0000.

2.2.2.2.3 Unsigned Integer

Unsigned integer numbers may be thought of as positive only. The unsigned numbers have nearly twice the magnitude of a signed number of the same length. Unsigned integer numbers lie in the following range:

$$0 \leq UI \leq [2^N-1]$$

The binary word is interpreted as having a binary point immediately to the right of the LSB. The most positive 16-bit unsigned integer is 65535 (\$FFFF). The most positive 32-bit unsigned integer is $2^{32}-1$ (\$FFFF FFFF). The smallest unsigned number is zero (\$0000).

If the extension bits are in use, the range is from zero to $+2^{40} - 1$.

Table 2-12. Two's Complement Word Representations

Signed Fractional		Signed Integer		Unsigned Integer	
\$7FFF	$1.0 - 2^{-15}$	\$7FFF	$2^{15} - 1$	\$FFFF	$2^{16} - 1$
				\$FFFE	$2^{16} - 2$
\$0001	2^{-15}	\$0001	+1		
\$0000	0	\$0000	0		
\$FFFF	-2^{-15}	\$FFFF	-1		
				\$0001	1
\$8000	-1.0	\$8000	-2^{15}	\$0000	0

2.2.2.3 Multiplication

Most of the operations are performed identically in fractional and integer arithmetic. However, the multiplication operation is not the same for integer and fractional arithmetic. As illustrated in Figure 2-4, fractional and integer multiplication differ by a 1-bit shift. Any binary multiplication of two N-bit signed numbers gives a signed result that is 2N-1 bits in length. This 2N-1 bit result must then be correctly placed into a field of 2N-bits to correctly fit into the on-chip registers. For correct fractional multiplication, an extra 0-bit is placed at the LSB to give a 2N-bit result. For correct integer multiplication, an extra sign bit is placed at the MSB to give a 2N-bit result.

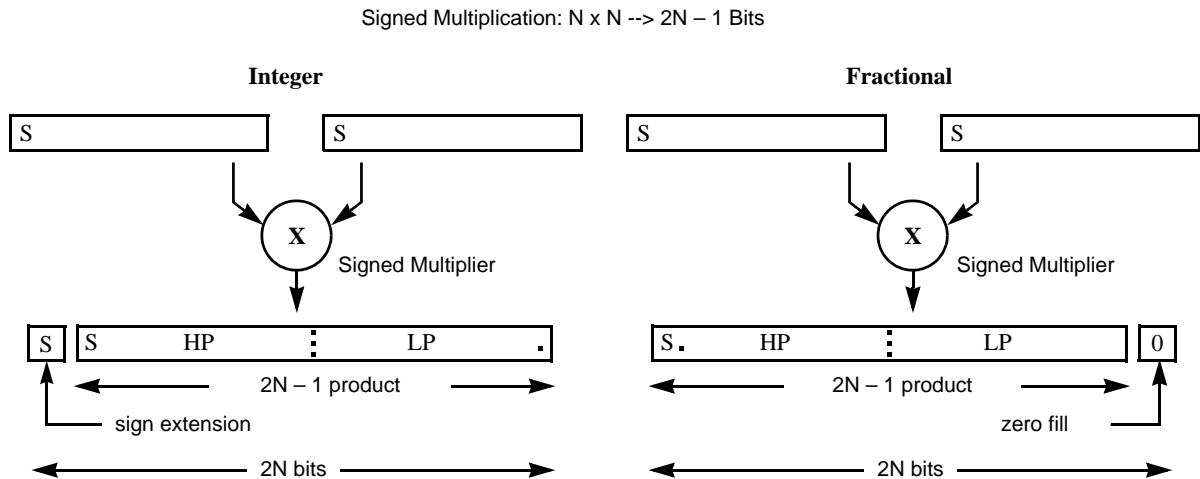


Figure 2-4. Fractional and Integer Multiplication

The MPY, MAC, MPYR, and MACR instructions perform fractional multiplication and fractional multiply-accumulation. The IMPY and the IMAC instructions perform integer multiplication.

2.2.2.4 Division

Fractional division of both positive and signed values is supported using the DIV instruction. The dividend (numerator) is a 32-bit fraction and the divisor (denominator) is a 16-bit fraction. For a detailed description of the DIV instruction, see [Appendix A, "SC140 DSP Core Instruction Set."](#)

2.2.2.5 Unsigned Arithmetic

Unsigned arithmetic can be performed on the SC140 core architecture. Most of the unsigned arithmetic instructions are performed the same as the signed instructions. However, some operations require special hardware and are implemented as separate instructions.

2.2.2.5.1 Unsigned Multiplication

Unsigned multiplication (MPYUU, MACUU) and mixed unsigned-signed multiplication (MPYSU, MACSU) are used to support double precision, as described in [Section 2.2.2.8, "Multi-Precision Arithmetic Support."](#) These instructions can be used for unsigned arithmetic multiplication.

2.2.2.5.2 Unsigned Comparison

When performing an unsigned comparison, the condition code computation is different from signed comparisons. The most significant bit of the unsigned operand has a positive weight, while in signed representation it has a negative weight. Special instructions are implemented to support unsigned comparison such as CMPHI (compare greater).

2.2.2.6 Rounding Modes

The SC140 DALU performs rounding of the full register to single precision if requested in the instruction. The high portion of the register is rounded according to the contents of the low portion of the register. Then the low portion is cleared. The boundary between the low portion and the high portion is determined by the scaling mode bits (S0 and S1) in the SR. Two types of rounding are implemented, convergent rounding and two's complement rounding. The type of rounding is selected by the rounding mode (RM) bit in the SR.

Table 2-13 shows the boundary between the high portion and the low portion depending on scaling. The scaling adjustment is disabled if arithmetic saturation mode is selected.

Table 2-13. Rounding Position in Relation to Scaling Mode

S1	S0	Scaling Mode	High Portion	Low Portion
0	0	No Scaling	39–16	15–0
0	1	Scale Down	39–17	16–0
1	0	Scale Up	39–15	14–0

2.2.2.6.1 Convergent Rounding

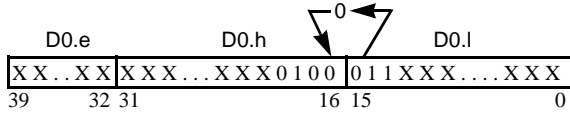
Convergent rounding (also called round-to-nearest even number) is the default rounding mode. It is selected when the rounding mode (RM) bit in the SR is cleared. The traditional rounding method rounds up any value greater than one-half, and rounds down any value less than one-half. However, the question arises as to which way one-half should be rounded. If it is always rounded one way, the results are eventually biased in that direction. Convergent rounding, however, removes the bias by rounding down if the high portion is even (LSB = 0) and rounding up if the high portion is odd (LSB = 1).

For no scaling, the higher portion (HP) of the register is bits 39:16; the low portion (LP) is bits 15:0. The HP is incremented by one bit if the LP was $> 1/2$, or if the LP = $1/2$ and bit 16 was 1 (odd). The HP is left alone if the LP was $< 1/2$, or if LP = $1/2$ and bit 16 was 0 (even). After rounding, the LP is cleared. If scaling down is selected, the HP is bits 39:17 and the LP is bits 16:0. If scaling up is selected, the HP is bits 39:15 and the LP is bits 14:0.

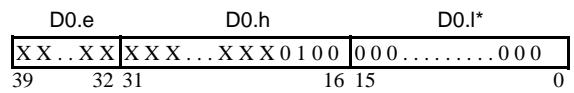
Figure 2-5 shows the four cases for rounding a number in the Dn.h register. If scaling is set in the SR, the rounding position is updated to reflect the alignment of the result when it is put on the data bus. However, the contents of the register are not scaled.

Case I: If $D0.I < \$8000$ ($1/2$), then round down (add nothing)

Before Rounding

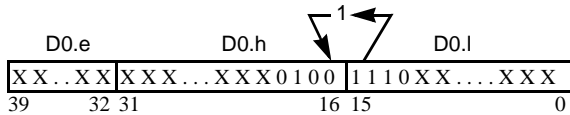


After Rounding

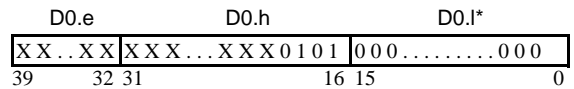


Case II: If $D0.I > \$8000$ ($1/2$), then round up (add 1 to D0.h)

Before Rounding

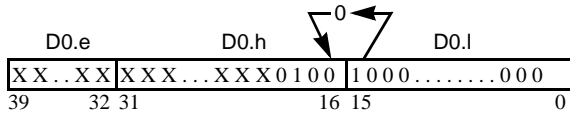


After Rounding

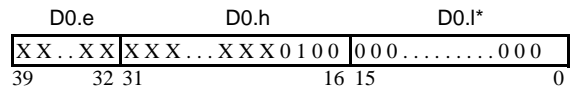


Case III: If $D0.I = \$8000$ ($1/2$), and the LSB of D0.h = 0, then round down (add nothing)

Before Rounding

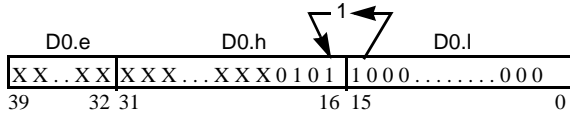


After Rounding

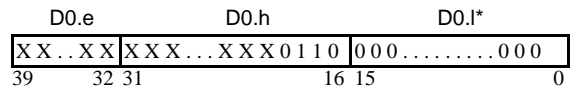


Case IV: If $D0.I = \$8000$ ($1/2$), and the LSB of D0.h = 1, then round up (add 1 to D0.h)

Before Rounding



After Rounding



*D0.I is always clear, performed during RND, MPYR, and MACR.

Figure 2-5. Convergent Rounding (No Scaling)



2.2.2.6.2 Two's Complement Rounding

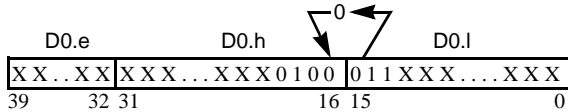
When two's complement rounding is selected by setting the rounding mode (RM) bit in the SR, all values greater than or equal to one-half are rounded up, and all values less than one-half are rounded down. Therefore, a small positive bias is introduced.

For no scaling, the higher portion (HP) of the register is bits 39:16; the low portion (LP) is bits 15:0. The HP is incremented by one bit if the LP was $\geq 1/2$. The HP is left alone if the LP was $< 1/2$. After rounding, the LP is cleared. If scaling down is selected, the HP is bits 39:17 and the LP is bits 16:0. If scaling up is selected, the HP is bits 39:15 and LP is bits 14:0.

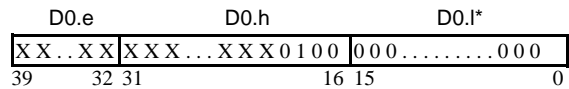
Figure 2-6 shows the four cases for rounding a number in the Dn.h register. If scaling is set in the SR, the rounding position is updated to reflect the alignment of the result when it is transferred to the data bus. However, the contents of the register are not scaled.

Case I: If $D0.l < \$8000$ ($1/2$), then round down (add nothing)

Before Rounding

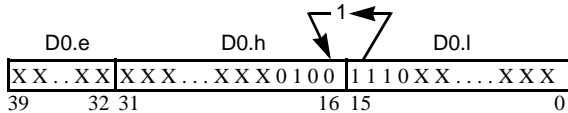


After Rounding

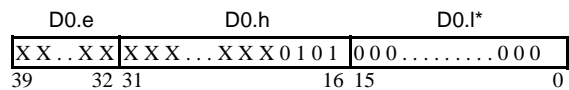


Case II: If $D0.l > \$8000$ ($1/2$), then round up (add 1 to D0.h)

Before Rounding

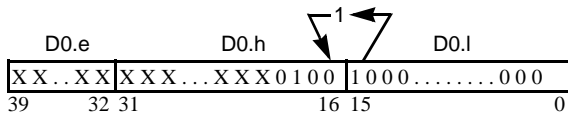


After Rounding

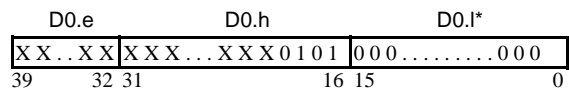


Case III: If $D0.l = \$8000$ ($1/2$), and the LSB of D0.h = 0, then round up (add 1 to D0.h)

Before Rounding

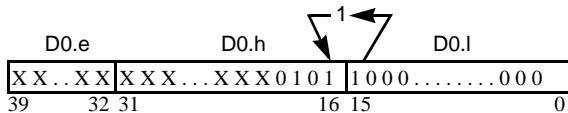


After Rounding

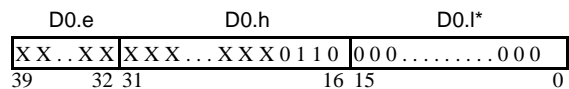


Case IV: If $D0.l = \$8000$ ($1/2$), and the LSB of D0.h = 1, then round up (add 1 to D0.h)

Before Rounding



After Rounding



*D0.l is always cleared, performed during RND, MPYR, and MACR.

Figure 2-6. Two's Complement Rounding (No Scaling)

2.2.2.7 Arithmetic Saturation Mode

By setting the arithmetic saturation mode (SM) bit in the SR, the arithmetic unit's result is limited to 32 bits (high portion and low portion). The dynamic range of the DALU is therefore reduced to 32 bits. The purpose of this bit is to provide a saturation mode for algorithms that do not recognize or cannot take advantage of the extension bits.

Arithmetic saturation operates by checking whether bits 39–31 of a relevant DALU instruction result in all ones or all zeros. If they are not, and if bit 39 is one, the result receives the negative saturation constant \$FF 8000 0000. If bit 39 is zero, the result receives the positive saturation constant \$00 7FFF FFFF. If saturation occurs, the DOVF bit in the EMR register is set.¹

The calculation for saturation is not affected by the scaling mode. In the same way, the rounding of the saturation constant during execution of MPYR, MACR and RND instructions is independent of the scaling mode: \$00 7FFF FFFF is rounded to \$00 7FFF 0000 and \$FF 8000 0000 is unchanged.

The instructions that are affected by arithmetic saturation mode are: MAC, MPY, MACR, MPYR, SUB, ADD, NEG, ABS, RND, INC, ADR, SBR, SUBL, ASR, SUBNC, ADDNC, and ASL.

When the arithmetic saturation mode is set, for most of the instructions, the scaling mode bits are ignored for the calculation of the Ln bit, and the Ln bit cannot be set. For instructions ADC, DIV, SBC, TFR, TFRT, and TFRF, however, the arithmetic saturation mode is ignored, and the Ln bit will be calculated. These six are dependent on arithmetic saturation mode to the extent that scaling is not considered in the Ln bit calculation if arithmetic saturation mode is on. See [Section 2.2.1.7, “Scaling and Arithmetic Saturation Mode Interactions,”](#) on page 2-16 for more information.

The arithmetic saturation mode is always disabled during the execution of the following instructions: TFR, TFRT, TFRF, MAX, MAXM, MIN, ADD2, SUB2, DIV, SBC, ADC, MAX2, MAX2VIT, DMACSU, DMACSS, MACSU, MACUS, MACUU, MPYSU, MPYUU, MPYUS, IADDNC, CMPHI, DECEQ, DECGE all integer multiplication operations, and all BFU operations as described in Table 2-6 on page 2-13. If the result of these instructions should be saturated, a SAT.L Dn instruction must be executed following the original instruction.

If the arithmetic saturation mode is set and data saturation occurs, the sticky data overflow bit (DOVF) in the EMR is set to signify that the arithmetic result before saturation cannot be represented in 32 bits. Note that if arithmetic saturation mode is not set, the DOVF bit is set when overflow from 40 bits occurs. Table 2-14 provides an example of the arithmetic saturation mode.

Table 2-14. Arithmetic Saturation Example

Instruction	Memory/ Register	New Value	Comments
bmset #0004,sr.l	sr	\$00e4 0004	Arithmetic saturation mode set
moveu.w #07ff,d0.h	d0	\$7ff 0000	d0.h set with the most positive 2's complement number
moveu.w #07ff,d1.h	d1	\$7ff 0000	d1.h set with the most positive 2's complement number
add d0,d1,d3	d3	\$0:00:7fff ffff	Max positive constant loaded in D3. L3 bit not set from overflow
	emr	\$0000 0004	DALU overflow bit set

1. In case of a 40-bit overflow which takes place in conjunction with arithmetic saturation, the constant being chosen is undefined, and it can be either the negative or positive constant.

2.2.2.8 Multi-Precision Arithmetic Support

The SC140 DALU supports multi-precision arithmetic for fractional and integer operations.

2.2.2.8.1 Fractional Multi-Precision Arithmetic

A set of DALU instructions is provided for fractional multi-precision multiplications. When these instructions are used, the multiplier accepts some combinations of two's complement signed and unsigned formats. Table 2-15 lists these instructions.

Table 2-15. Fractional Signed and Unsigned Two's Complement Multiplication

Instruction	Description
MPYSU/MACSU	Fractional multiplication and multiply-accumulate with signed \times unsigned operands
MPYUS/MACUS	Fractional multiplication and multiply-accumulate with unsigned \times signed operands
MPYUU/MACUU	Fractional multiplication and multiply-accumulate with unsigned \times unsigned operands
DMACSS	Fractional multiplication with signed \times signed operands and 16-bit arithmetic right shift of the accumulator before accumulation
DMACSU	Fractional multiplication with signed \times unsigned operands and 16-bit arithmetic right shift of the accumulator before accumulation

Figure 2-7 shows how the DMAC instruction is implemented.

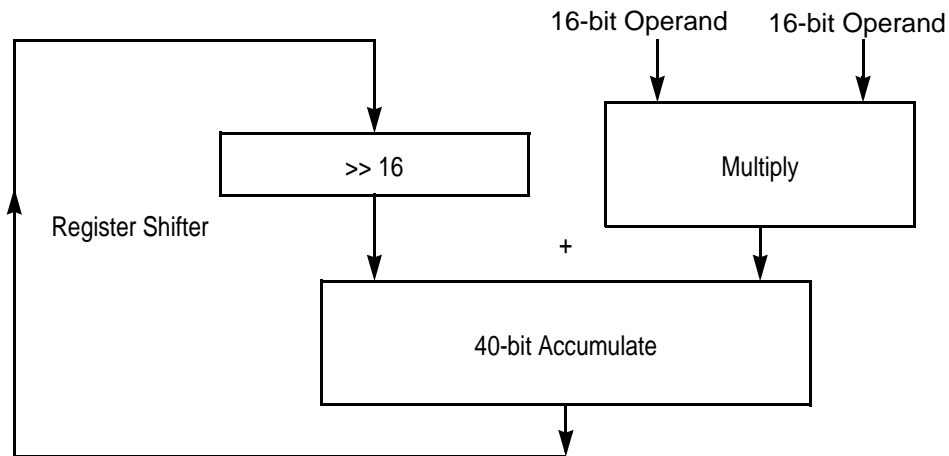


Figure 2-7. DMAC Implementation

Figure 2-8 illustrates the use of these instructions in the case of a double-precision multiplication of 32-bit x 32-bit operands. The “Unsigned x Unsigned” operation is used to multiply or multiply-accumulate the unsigned low portion of one double-precision number with the unsigned low portion of the other double-precision number. The “Signed x Unsigned” and “Unsigned x Signed” operations are used to multiply or multiply-accumulate the signed high portion of one double-precision number with the unsigned low portion of the other double-precision number. The “Signed x Signed” operation is used to multiply or multiply-accumulate the two signed high portions of two signed double-precision numbers. The TFRx instructions in parentheses are optional instructions that are used only in case all 64 bits of the result are needed. Otherwise, the result is truncated to a 32-bit fraction.

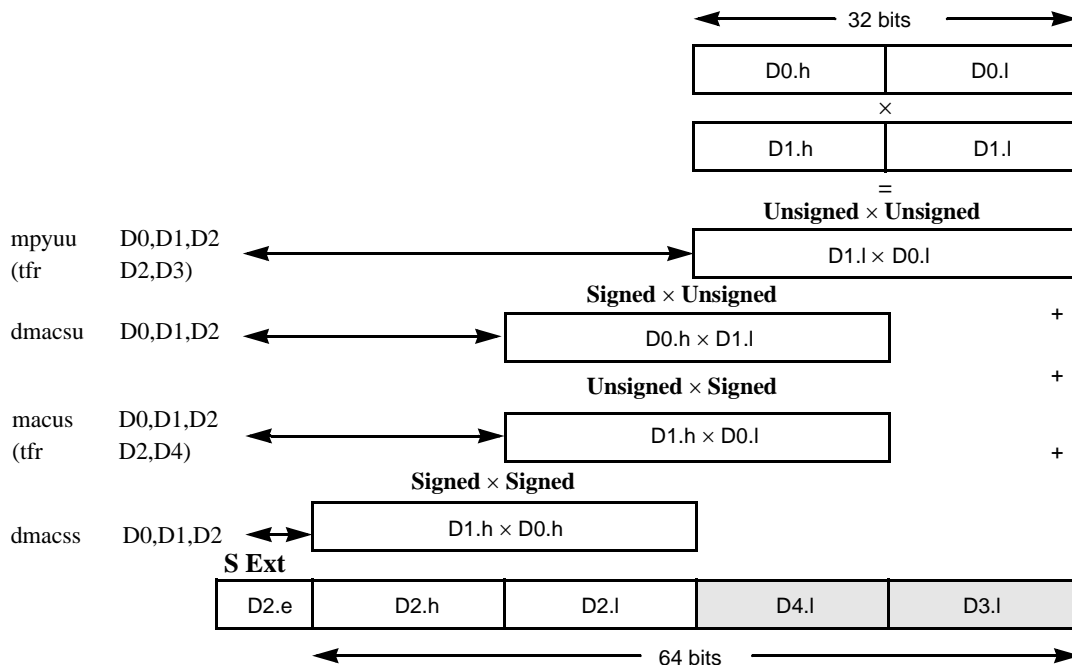


Figure 2-8. Fractional Double-Precision Multiplication

Figure 2-9 illustrates the use of the fractional multiplication and multiply-accumulate instructions in the case of a mixed double-precision multiplication of 16-bit by 32-bit signed operands. The “Signed x Unsigned” operation is used to multiply the signed high portion of one single-precision number with the unsigned low portion of the other double-precision number. The “Signed x Signed” DMAC operation is used to multiply-accumulate the two signed high portions of the two signed operands. The TFRx instruction in parentheses is an optional instruction that is used only in case all 48 bits of the result are needed. Otherwise, the result is truncated to a 32 bit fraction.

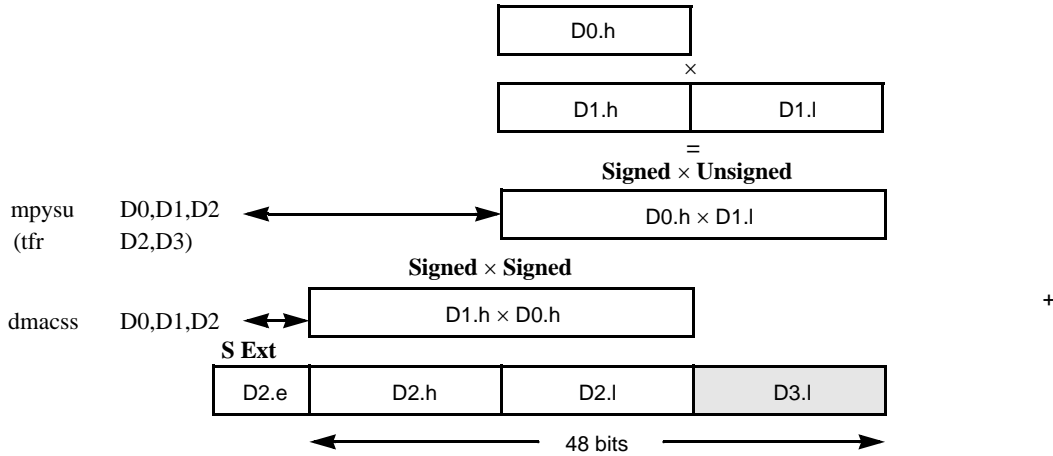


Figure 2-9. Fractional Mixed-Precision Multiplication

2.2.2.8.2 Integer Multi-Precision Arithmetic

A set of DALU operations is provided for integer multi-precision multiplications. When these instructions are used, the multiplier accepts some combinations of two’s complement signed and unsigned formats. Both signed and unsigned multi-precision multiplication are supported. Table 2-16 lists these instructions.

Table 2-16. Integer Signed and Unsigned Two’s Complement Multiplication

Instruction	Description
IMPYSU/IMACSU	Integer multiplication and multiply-accumulate with signed x unsigned operands
IMPYUU	Integer multiplication with unsigned x unsigned operands
IMPYHLUU	Integer multiply unsigned x unsigned: first source from high portion, second from low portion
IMACLHUU	Integer multiply-accumulate unsigned x unsigned: first source from low portion, second from high portion

Figure 2-10 illustrates the use of these instructions in the case of a signed integer double-precision multiplication of 32-bit by 32-bit signed operands. In this example, only a 32-bit result is generated. The most significant 32 bits are shifted out. The “Unsigned x Unsigned” operation is used to multiply or multiply-accumulate the unsigned low portion of one double-precision number with the unsigned low portion of the other double-precision number. The “Signed x Unsigned” and “Unsigned x Signed” operations are used to multiply or multiply-accumulate the signed high portion of one double-precision number with the unsigned low portion of the other double-precision number. This example generates only a 32-bit integer.

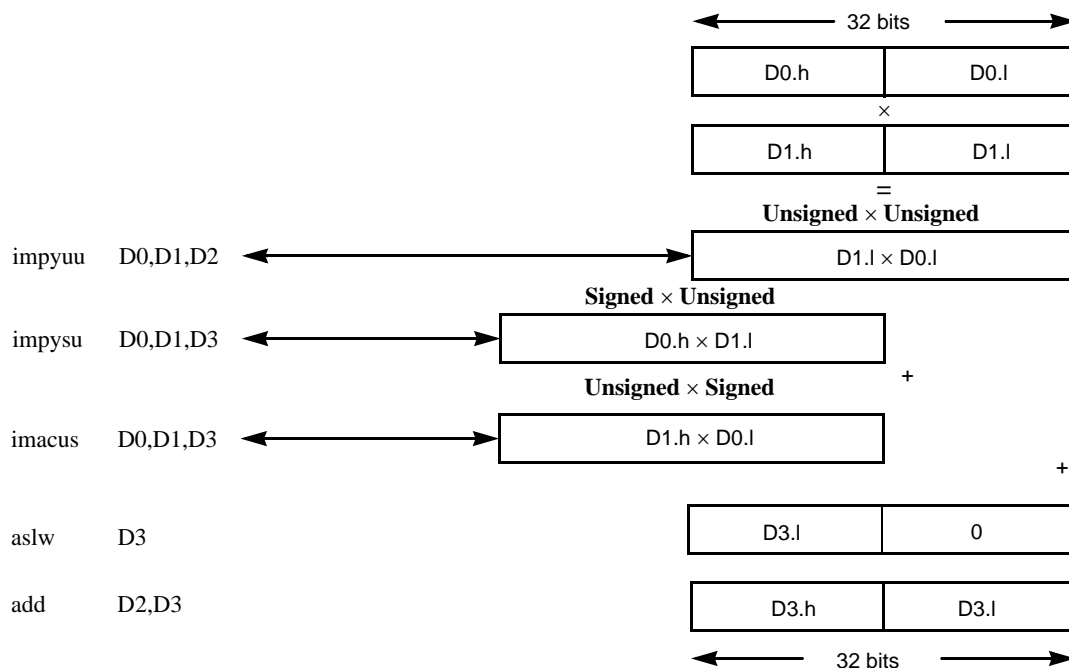


Figure 2-10. Signed Integer Double-Precision Multiplication

Figure 2-11 illustrates the use of these instructions in the case of an unsigned integer double-precision multiplication of 32-bit by 32-bit unsigned operands. In this example, only a 32-bit result is generated. The most significant 32-bits are shifted out. All multiplications are of the “Unsigned x Unsigned” type using different combinations of high and low portions.

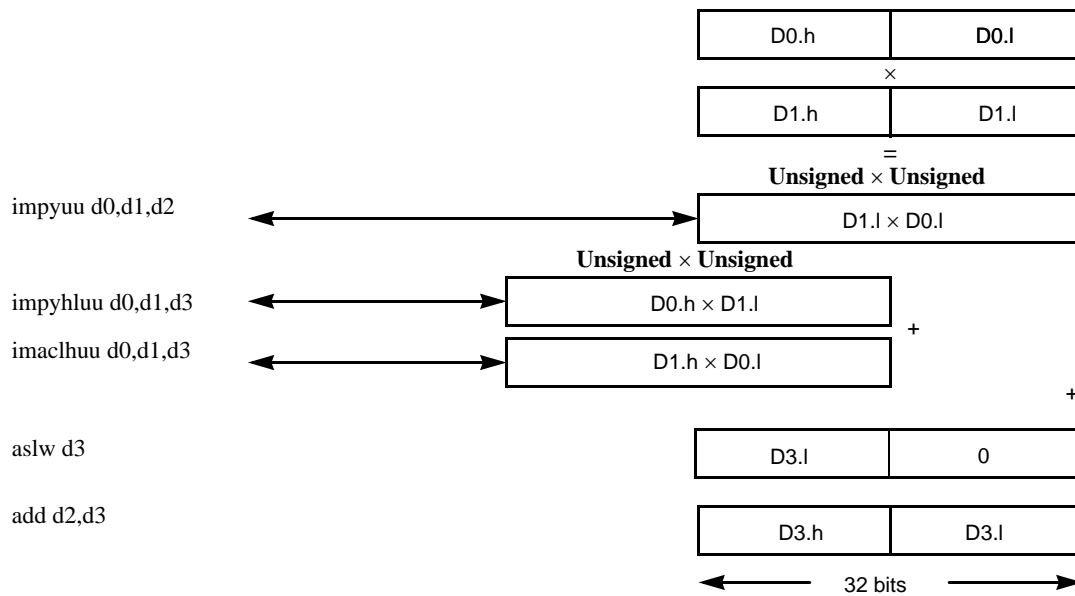


Figure 2-11. Unsigned Integer Double-Precision Multiplication

2.2.2.9 Viterbi Decoding Support

A set of DALU and AGU operations is provided for Viterbi decoding kernels. A special MAX2VIT operation is defined. This instruction functions as a regular MAX2 instruction and is used to transfer two 16-bit maximum signed values. In addition, the MAX2VIT instruction updates two Viterbi flags (VFs) which reside in the status register as described in [Section 3.1.1, “Status Register \(SR\),”](#) on page 3-1. Complementary AGU move operations are provided (VSL instructions). For a full description of the Viterbi instructions, see [Appendix A, “Viterbi Shift Left Move \(AGU\) VSL,”](#) on page A-422.

2.3 Address Generation Unit

The AGU is one of the execution units in the SC140 core. The AGU performs effective address calculations using the integer arithmetic necessary to address data operands in memory. It also contains the registers used to generate the addresses. The AGU implements four types of arithmetic: linear, modulo, multiple wrap-around modulo, and reverse-carry. It operates in parallel with other chip resources to minimize address generation overhead. The AGU also generates change-of-flow program addresses as well as updates the stack pointer (SP), whenever needed.

2.3.1 AGU Architecture

The major components of the AGU are listed below:

- Eight low bank address registers (R0–R7)
- Eight high bank address registers (R8–R15), or alternatively, eight base address registers (B0–B7)
- Two stack pointers (NSP, ESP), only one of which is active at a time (SP)
- Four offset registers (N0–N3)
- Four modifier registers (M0–M3)
- A modifier control register (MCTL)
- Two address arithmetic units (AAU)
- One bit mask unit (BMU)

In this section, the registers are referred to as:

- R_n for any of the R0–R15 address registers
- B_n for any of the B0–B7 base address registers
- N_i for any of the N0–N3 offset registers
- M_j for any of the M0–M3 modifier registers

All the R_n , B_n , SP, N_i , and M_j registers are referred to as AGU registers. All of the AGU registers are 32-bits.

Figure 2-12 shows a block diagram of the AGU.

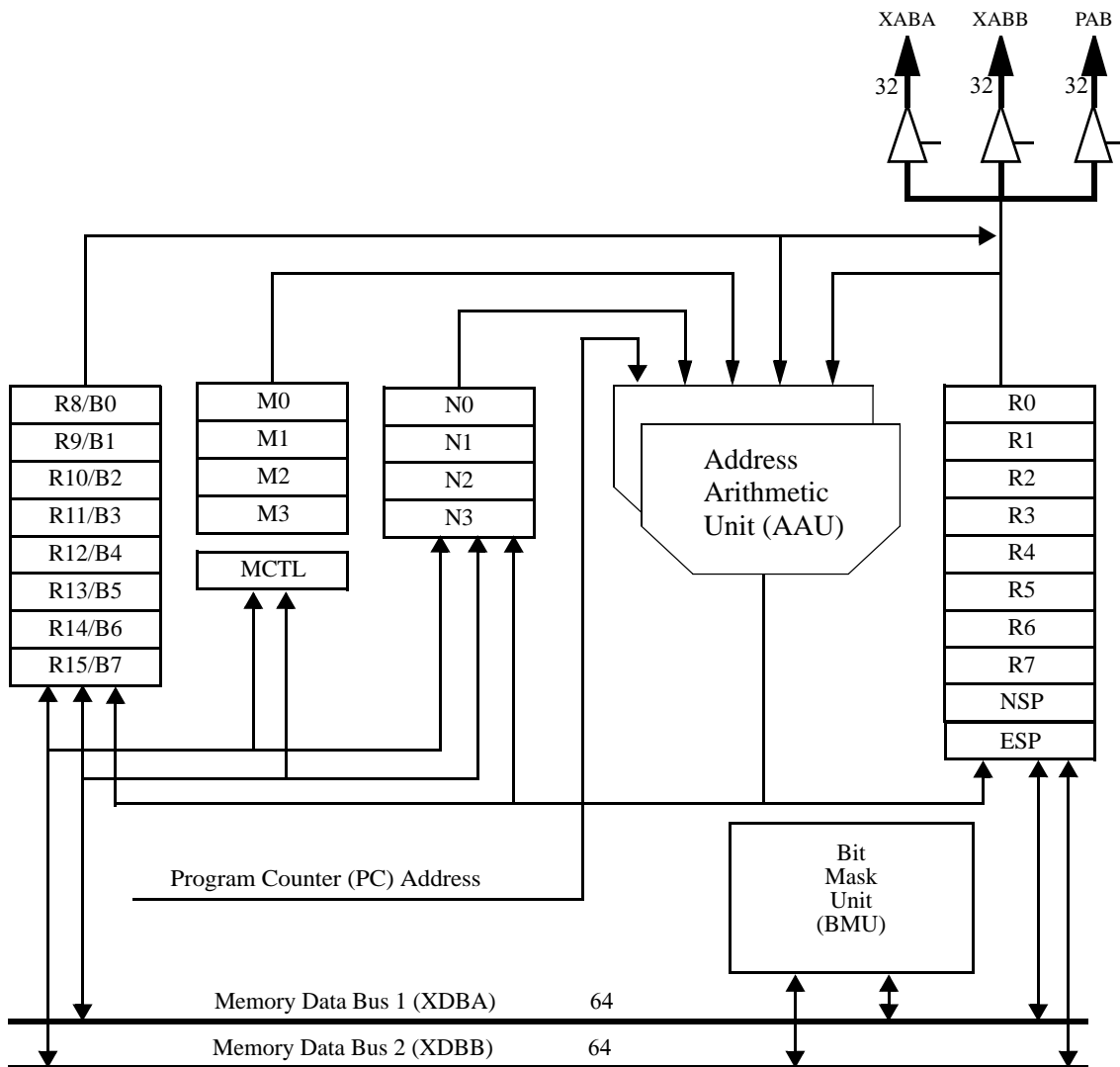


Figure 2-12. AGU Block Diagram

All sixteen address registers (R0–R15) as well as the NSP or ESP are used for generating addresses in the register indirect addressing modes. All four offset registers (N0–N3) can be used by all sixteen address registers. The four modifier registers (M0–M3) can only be used by the low bank of eight address registers (R0–R7).

The base address (Bn) registers are uniquely associated with the low bank of Rn registers such that B0 is used with R0, B1 with R1, and so on.

The BMU is used to perform bit mask operations such as setting, clearing, changing, or testing bits in a destination according to an immediate mask operand. Data is loaded into the BMU over the data memory buses XDBA or XDBB. The result is written back over XDBA or XDBB to the destinations in the next cycle. All bit mask instructions are typically executed in two cycles and work on 16-bit data. This data can be a memory location or a portion (high or low) of a register. For more information, see [Section 2.3.6, “Bit Mask Instructions.”](#)

During every instruction cycle, the two AAUs can generate one 32-bit program memory address on the PAB (in case of change of flow) or two 32-bit data memory addresses (one on each of the XABA and XABB). Each AAU can generate an address to access a byte, a 16-bit word, a 32-bit long word, or a 64-bit two-word long operand in memory to feed into the DALU in a single cycle.

Each AAU can update one address register during one instruction cycle. The modifier control register (MCTL) specifies the type of arithmetic to be used in the address register update calculation. The address arithmetic instructions provide arithmetic operations for address calculations or for general purpose calculations.

The two AAUs are identical. Each contains a 32-bit full adder, called an offset adder, which can perform the following:

- Add or subtract two AGU registers
- Add an immediate value
- Increment or decrement an AGU register
- Add the PC
- Add with reverse-carry

The offset adder can also perform compare or test operations as well as arithmetic and logical shifts. The offset values added in this adder can be pre-shifted left by 1, 2, or 3 bits according to the access width. In reverse-carry mode, the carry propagates in the opposite direction.

A second full adder, called a modulo adder, adds the summed result of the first full adder to a modulo value, M or minus M , where M is stored in the selected modifier register. In modulo mode, a modulo comparator tests whether the result is inside the buffer by comparing the results to the B register, choosing the correct result from the offset adder or the modulo adder.

For more information, see [Section 2.3.5, “Arithmetic Instructions on Address Registers.”](#)

2.3.2 AGU Programming Model

The programming model of the AGU is shown in Figure 2-13.

The address registers can be programmed for linear addressing, modulo addressing (regular or multiple wrap-around), and reverse-carry addressing. Automatic updating of address registers is available when using address register indirect addressing.

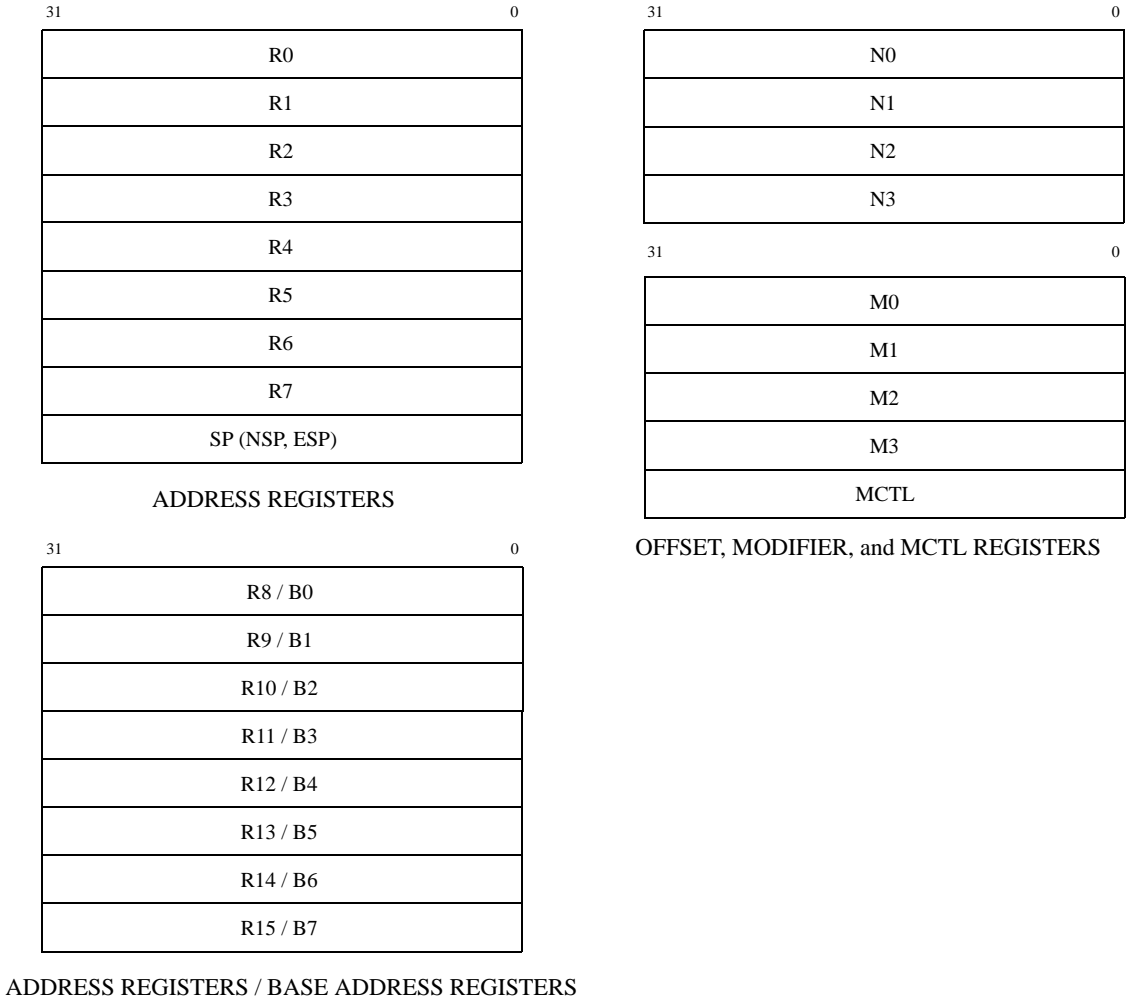


Figure 2-13. AGU Programming Model

2.3.2.1 Address Registers (R0–R15)

The sixteen 32-bit address registers R0–R15 can contain addresses or general-purpose data. These are 32-bit read/write registers. The 32-bit address in a selected address register is used in calculating the effective address of an operand. The contents of an address register can point directly to data, or can be used as an index.

The sixteen address registers R0–R15 are composed of two separate banks, a low bank (R0–R7) and a high bank (R8–R15). The high bank can be used alternatively as a base address register bank (B0–B7). Each address register R_n of the high bank can serve as an address register on condition that the corresponding B_{n-8} register is not used. Both R_n and B_{n-8} are mapped to the same physical register. For example, R8 is available only if R0 is not being used in modulo addressing since this requires the base address register B0.

Use of both R_n and B_{n-8} notations as source and destination of move-like instructions is permitted, regardless of the use of the physical register as Base modulo or as a pointer. For example:

```
MOVE.L #ADDRESS, B0
...
MOVE.W (R8), D0
```

See [Section 2.3.2.6, “Modifier Control Register \(MCTL\),”](#) for further information. The high bank of registers can only be used as pointers in the linear mode of addressing since the other modes of addressing are only encoded for the low bank in the MCTL register.

In addition, an address register can be post-updated according to the addressing mode selected. If an address register is updated, one of the modifier registers (M_j) can be used to specify the type of update arithmetic. Offset registers (N_i) are used for post-incrementing and indexing by offset.

The address register modification can be performed by either of the two AAUs. Most addressing modes modify the selected address register in a read-modify-write fashion. The address register is read, its contents are modified by the associated modulo arithmetic unit, and the register is written with the appropriate output of the AAU. The form of address register modification performed by the address arithmetic unit is controlled by the contents of the offset and modifier registers described in the following sections.

2.3.2.2 Stack Pointer Registers (NSP, ESP)

The SC140 core has two stack pointer registers: the normal stack pointer (NSP) and the exception stack pointer (ESP). These 32-bit registers are used implicitly in all PUSH and POP instructions. Only one stack pointer is active at one time according to the mode:

- In Normal working mode, the NSP is used.
- In Exception working mode, the ESP is used.

The EXP bit in the status register (SR) determines the active working mode. The active stack pointer (SP) is used explicitly for memory references when used with the address register indirect modes. The stack pointers point to the next unoccupied location in the stacks. They are post-incremented on all the implicit PUSH operations and pre-decremented on all the implicit POP operations.

Note: Both stack pointer registers must be initialized explicitly by the programmer after reset.



2.3.2.2.1 Shadow Stack Pointer Registers

Both stack pointers have shadow registers which contain a decremented value of the stack pointers. When the shadow register is not valid, the POP instruction is executed in two cycles. The first cycle is used to decrement the stack pointer. When the shadow register is valid, the POP instruction is executed in only one cycle.

When an SP is written by the AAU register transfer (TFRA), its shadow register automatically becomes invalid. When a PUSH/POP instruction is executed, the shadow register of the active SP becomes valid. As a result, during consecutive POPs, even in the worst case, only the first POP requires an additional cycle.

2.3.2.2.2 Initializing ESP

ESP should be initialized using the AAU register transfer (TFRA) instruction. This guarantees a valid ESP value even if execution of this instruction is interrupted by an exception. The TFRA instruction is considered an address arithmetic operation. The ESP is updated at the address generation pipeline stage, avoiding pipeline conflicts.

2.3.2.3 Offset Registers (N0–N3)

The four 32-bit read/write offset registers N0–N3 can contain offset values used to increment or decrement address registers in address register update calculations. These registers can also be used for 32-bit general purpose storage. For example, the contents of an offset register can specify the offset into a table or the base of the table for indexed addressing, or can be used to step through a table at a specified rate (for example, five locations per step for waveform generation). Each address register can be used with each offset register. For example, R0 can be used with N0, N1, N2, or N3 for offset address calculations. The signed value in an offset register is pre-shifted to the left by 0, 1, 2, or 3 bits to align to the access width.

2.3.2.4 Base Address Registers (B0–B7)

The eight 32-bit read/write base address registers B0–B7 are used in modulo calculations. Each B register is associated with an R register (B0 with R0, and so on). When activating the modulo addressing mode, the B register contains the lower boundary value of the modulo buffer. The upper boundary of the modulo buffer is calculated by $B+M-1$, where M is the modifier register associated with the R register by MCTL.

When not used for modulo addressing, these registers can be used as high bank address registers (R8–R15). Both R_n and $B_{n,8}$ share the same physical register. For example, if R0 is not programmed for modulo addressing, the base address register B0 can serve as an additional address register R8.

2.3.2.5 Modifier Registers (M0–M3)

The four 32-bit read/write modifier registers M0–M3 can contain the value of the modulus modifier. These registers can also be used for general-purpose storage. When activating the modulo arithmetic, the contents of M_j specify the modulus. Each low address register can be used with each modifier register as programmed in the MCTL register.

2.3.2.6 Modifier Control Register (MCTL)

The MCTL register is a 32-bit read/write register. This control register is used to program the address mode (AM) for each of the eight low address registers (R0–R7). The addressing mode of the high address register file (R8–R15) cannot be programmed and functions in linear addressing mode only. The format of MCTL is shown in Figure 2-14.

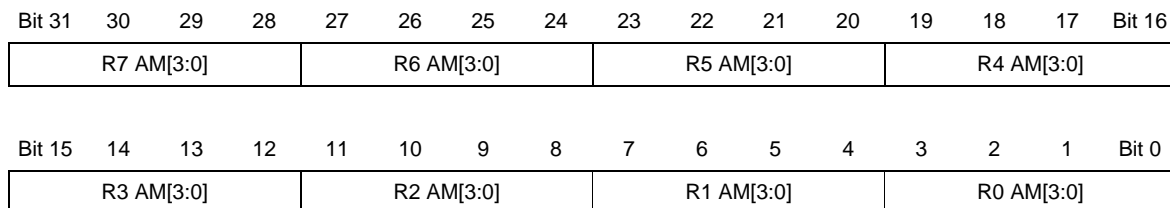


Figure 2-14. Modifier Control Register (MCTL) Format

The AM bits (AM3, AM2, AM1, AM0) associated with each address register (R0-R7) reflect the address modifier mode of this address register as shown in Table 2-17. Each of the R_n registers can use M0, M1, M2, or M3 as their associated modulo register either in modulo addressing mode, or in multiple wrap-around modulo addressing mode. When activating the modulo addressing mode, the corresponding B register is used to define the lower boundary value (B0 with R0, and so on). The linear or the reverse-carry addressing modes can also be used, freeing the B register to be used as an additional linear address register.

The high bank of the address register file (R8–R15) can only be used in linear addressing mode. Each R_n (n = 8:15) is available only if the corresponding B_{n-8} register is not used since both R_n and B_{n-8} are mapped to the same physical register.

Table 2-17. Address Modifier (AM) Bits

AM3	AM2	AM1	AM0	Address Modifier Modes
0	0	0	0	Linear addressing
0	0	0	1	Reverse-carry addressing
1	0	0	0	M0 used—Modulo addressing
1	0	0	1	M1 used—Modulo addressing
1	0	1	0	M2 used—Modulo addressing
1	0	1	1	M3 used—Modulo addressing
1	1	0	0	M0 used—Multiple wrap-around modulo addressing
1	1	0	1	M1 used—Multiple wrap-around modulo addressing
1	1	1	0	M2 used—Multiple wrap-around modulo addressing
1	1	1	1	M3 used—Multiple wrap-around modulo addressing

MCTL is initialized to zero at reset, setting a default linear mode for all R_n registers. All other AM field combinations are reserved and should not be used.



2.3.3 Addressing Modes

The SC140 core provides four types of addressing modes:

- Register direct
- Address register indirect
- PC relative
- Special

The addressing modes are related to where the operands are to be found and how the address calculations are to be made. These modes are described in the following sections:

2.3.3.1 Register Direct Modes

The register direct addressing modes specify that the operand is in one or more of the DALU registers, AGU registers, or control registers, and are classified as register references.

- **Data or Control Register Direct** — The operand is in one, two, or four DALU registers as specified in a portion of the data bus movement field in the instruction. An example is: `mac d4, d5, d6`, which uses data registers d4, d5, and d6 as sources for the multiply-accumulate operation. This addressing mode is also used to specify a control register operand for special instructions.
- **Address Register Direct** — The operand is in one of the twenty-seven AGU registers (R0–R7, R8–R15/B0–B7, N0–N3, M0–M3, MCTL, N/ESP) specified by a field in the instruction. An example is `addl1a r0, r1`, which performs a 1-bit arithmetic left shift on the data in R0, and adds the result to the data in R1.

2.3.3.2 Address Register Indirect Modes

The address register indirect modes specify that the address register is used to point to a memory location. The term indirect is used because the register contents are not the operand itself, but rather the operand address. These addressing modes specify that an operand is in a memory location and specify the effective address of that operand. These references are classified as memory references. The term “index” refers to an offset stored in a register. The term “displacement” refers to an offset from an immediate in the instruction.

- **No Update, (Rn)** — The operand address is in the address register. The contents of the address register are unchanged by executing the instruction. For R0–R7, the contents of the modifier control register (MCTL) are ignored. An example is: `bmc1r.w #004f, (r4)`. A word is read from memory location stored in r4, operated on, and written back to the same location. The address in r4 is unchanged.
- **Post-increment, (Rn)+** — The operand address is in the address register. After the operand address is used, it is incremented by the access width (1, 2, 4, or 8 bytes) and stored in the same address register. The access width is the number of bytes used by the active instruction on the memory data bus. Incrementing the operand address by the access width places the next available byte address in the register. The type of arithmetic used for updating R0–R7 is determined by programming the MCTL register. An example is: `move.f (r3)+, d2`. The data in the location identified by the value in r3 is moved to data register d2. Then the value in r3 is incremented by two.

- Post-decrement, (Rn)-** — The operand address is in the address register. After the operand address is used, it is decremented by the access width (1, 2, 4, or 8 bytes) and stored in the same address register. The type of arithmetic used for updating R0-R7 is determined by programming the MCTL register. An example is: `move.l (r3)-, d2`. In this case, the value in r3 is decremented by four after the move has taken place.
- Post-increment by Offset Ni, (Rn) + Ni** — The operand address is in the address register. After the operand address is used, it is incremented or decremented by an amount determined by the signed contents of the Ni register pre-shifted to the left by 0, 1, 2, or 3 bits according to the access width. The result is stored in the same address register. The type of arithmetic used for updating R0-R7 is determined by programming the MCTL register. The contents of the Ni register are unchanged. An example is: `move.w d3, (r2)+n3`. The access width is two, so the increment is twice the value in the n3 register.
- Indexed By Offset N0, (Rn + N0)** — The operand address is the sum of the contents of the address register and the signed contents of the N0 register, pre-shifted to the left by 0, 1, 2, or 3 bits according to the access width. The type of arithmetic used for updating R0-R7 is determined by programming the MCTL register. The contents of the Rn and N0 registers are unchanged. For example: `move.b d6, (r3+n0)`. The access width is one, so the contents of the n0 register are used directly to modify the address before the move is done.

Note that only the N0 offset register can be used in this addressing mode.

- Indexed by Address Register Rm, (Rn + Rm)** — The operand address is the sum of the contents of the address register Rn and the contents of the address register Rm, pre-shifted to the left by 0, 1, 2, or 3 bits according to the access width. The type of arithmetic used for updating R0-R7 is determined by programming the MCTL register. The contents of the Rn and Rm registers are unchanged. An example is: `move.l (r0+r2), d6`. Here, the access width is four, so the value in r2 is shifted left two bits before adding to the address in r0.

Note that only address registers (R0–R7) can be used as Rm.

- Short Displacement, (Rn + x)** — The operand address is the sum of the contents of the address register Rn and a short displacement x that occupies three bits in the instruction word. The displacement (unsigned) is first shifted to the left by 0, 1, 2, or 3 bits according to the access width. It is then zero-extended to 32 bits and added to Rn to obtain the operand address. Thus, the displacement can range from [0] to [+7] bytes, words, long words, or two long words according to the access width. The contents of the Rn register are unchanged. The type of arithmetic used for updating R0-R7 is determined by programming the MCTL register. An example is: `move.l d4, (r3+$1c)`. The access width is four, and the displacement encoded in the instruction is seven ($4 \times 7 = 28 = \$1c$).
- Word Displacement, (Rn + xxxx)** — The operand address is the sum of the contents of the address register Rn and an immediate displacement. The displacement is a signed 15-bit word that requires a second instruction word. It is sign-extended to 32 bits and then added to Rn to obtain the operand address. Thus, the displacement can range from [-16,384] to [+16,383] bytes, [-8192] to [+8191] words, [-4096] to [+4095] long words, or [-2048] to [+2047] two long words according to the access width. The contents of the Rn register are unchanged. The type of arithmetic used for updating R0-R7 is determined by programming the MCTL register.
- SP Short Displacement, (SP – xx)** — The instruction word contains a 5-bit or 6-bit short unsigned immediate index field. This field is first shifted to the left by 1 or 2 bits according to the access width, then zero-extended to form a 32-bit offset and subtracted from the active stack pointer (NSP in Normal mode, ESP in Exception mode) to obtain the operand address. Thus, the displacement can range from [0] to [31/63] words or long words according to the access width. The contents of the



active SP register are unchanged. The type of arithmetic used is always linear. An example is: `move.w #$ffff, (sp-$3e)`. The encoded displacement is 31, the maximum value of five bits, and the actual displacement is 62 (\$3e), since the access width is two.

- **SP Word Displacement, (SP + xxxx)**—The operand address is the sum of the contents of the active stack pointer (SP) and an immediate displacement. The displacement is a signed 15-bit word that requires a second instruction word. It is sign-extended to 32 bits and added to the active stack pointer (NSP in Normal mode, ESP in Exception mode) to obtain the operand address. Thus, the displacement can range from [-16,384] to [+16,383] bytes, [-8192] to [+8191] words, [-4096] to [+4095] long words, or [-2048] to [+2047] two long words according to the access width. The contents of the active SP register are unchanged. The type of arithmetic used is always linear. An example is: `move.l (sp+$2000), d2.e`. Here, the positive value \$2000 is added to the active stack pointer before the memory access.

2.3.3.3 PC Relative Mode

The PC relative address mode is used to calculate the program destination of change-of-flow instructions such as branches (BRA). In the PC relative addressing mode, the instruction encoding contains a signed displacement operand. The operand address is obtained by left-shifting (multiplying by two) the displacement and adding the result to the value of the program counter (PC). The operand is left-shifted because the addresses of the program instructions are word-aligned, and memory addressing is in units of bytes. The arithmetic used is always linear. For example, `bra _label2`. Assume that PC=\$0010 and that `_label2` is at location \$0020. The encoded displacement will be $(\$0020 - \$0010)/2 = \$0008$.

The number of bits occupied by the displacement in the instruction differs with the different kinds of PC relative instructions. In all cases, the displacement is first sign-extended to 32 bits, then multiplied by two, and added to the PC to obtain the operand address.

In the one-word conditional branch instructions, the displacement occupies 8 bits of the instruction word and can range from [-256] to [254] words. In the one-word unconditional branch instructions, the displacement occupies 10 bits of the instruction word and can range from [-1024] to [1022] words. In the two-word branch instructions, the displacement occupies 20 bits and can range from [-1,048,576] to [1,048,574] words. In the DOSETUP instruction, the displacement occupies 16 bits of the instruction. The displacement for the start address (SA) can range from [-65,536] to [65,534] words.

2.3.3.4 Special Addressing Modes

The special addressing modes do not use an address register when specifying an effective address. They either use an immediate value that is included in the instruction for the data value, such as the data value address, or they use a register that is implicitly referenced by the instruction for the data value.

- **Immediate Short Data** — A 5-bit, 6-bit, or 7-bit operand is part of the instruction operation word. The 5-bit zero-extended operand is used for DALU and AGU arithmetic instructions. The 6-bit zero-extended operand is used for DALU instructions to move short immediate data to an LCn register. The 7-bit sign-extended operand is used for immediate moves to a register. This reference is classified as a program reference. An example is: `doen2 #3f`. The value \$3f, 63, is loaded to loop counter 2.
- **Immediate Word Data** — This addressing mode requires a one-word instruction extension. The immediate data is a 16-bit operand. This reference is classified as a program reference. An example is: `doen2 #40`. The value 64 is loaded to loop counter 2. The value exceeds the 6-bit limit for immediate short data, so an extra word is needed for the encoding.
- **Immediate Long Data** — This addressing mode requires a two-word instruction extension. The immediate data is a 32-bit operand. This reference is classified as a program reference. An example is: `move.l #f00d0d01, n0`. The 32-bit unsigned value is moved to the general register n0.
- **Absolute Word Address** — This addressing mode requires a one-word instruction extension. The operand address occupies 16 bits in the instruction operation words, and is zero-extended to form a 32-bit address. This reference is classified as a memory reference. An example is: `move.w ($8a20), d0`.
- **Absolute Long Address** — This addressing mode requires a two-word instruction extension. A 32-bit address is contained in the instruction words. This reference is classified as a memory reference. An example is: `move.w ($34008a20), d0`.
- **Absolute Jump Address** — The operand occupies 32 bits in the instruction operation words. It requires a two-word instruction extension. This reference is classified as a program reference. An example is: `jmp 1b14`, where the instruction is encoded with the program memory address of 1b14.
- **Implicit Reference** — Some instructions make implicit reference to the PC, normal or exception stack, loop registers (SA0, SA1, SA2, SA3, LC0, LC1, LC2, LC3), or status register (SR). These registers are implied by the instruction, and their use is defined by the individual instruction descriptions. An example is: `tfra osp, r2`, which transfers the 32-bit word stored at the other (non-active) stack pointer to address register R2.



2.3.3.5 Memory Access Width

The SC140 core supports variable width access to data memory. With every memory access, the core sends one of four signals to the memory interface to designate whether the access width is 8 bits, 16 bits, 32 bits, or 64 bits wide. The access width is determined by the type of MOVE instruction being used. For example, MOVE.B is used for byte access. MOVE.W is used for word access. For long-word access, MOVE.L, MOVE.2F, and MOVE.2W are used. And, for two long-word access, MOVE.2L, MOVE.4F, and MOVE.4W are used.

The memory addresses are always in units of bytes. For example, addresses for two-word MOVE operations to/from memory are available in multiples of four in order to best align the data with the byte addressing.

Address calculations and register update calculations are performed according to the memory access width as shown in Table 2-18.

Table 2-18. Access Width Support for Address and Register Update Calculations

Addressing Mode	Calculation	Memory Access Width			
		Byte	Word	Long	Two Long
Post-increment (Rn) + Post-decrement (Rn) -	Rn register post-increment or post-decrement by —>	1	2	4	8
Post-increment by Offset (Rn)+Ni	Rn register post-increment by ->	Ni*1	Ni*2	Ni*4	Ni*8
Indexed by Offset N0 (Rn + N0)	Actual address offset	N0	2*N0	4*N0	8*N0
Indexed by Address Register Rm (Rn + Rm)	Actual address offset	Rm	2*Rm	4*Rm	8*Rm
Short Displacement (Rn + x)	Actual address displacement	x	x	x	x
Word Displacement (Rn + xxxx)	Actual address displacement	xxxx	xxxx	xxxx	xxxx
SP update in Push/Pop	SP post-increment or pre-decrement by —>	8	8	8	8
SP Short Displacement (SP - xx)	Actual address displacement	NA	xx	xx	NA
SP Word Displacement	Actual address displacement	xxxx	xxxx	xxxx	xxxx

2.3.3.6 Memory Access Misalignment

Each access to the memory generated by the core should be aligned according to the access type. If the alignment rule is violated, erroneous data may be fetched from the memory. In addition, an exception may be generated to identify that an unaligned access occurred. For more information, see [Section 5.8, “Exception Processing,”](#) on page 5-46.

Table 2-19 summarizes the memory address alignment rule for each type of memory access.

Table 2-19. Memory Address Alignment

Access Type	Aligned Address
Byte access	Any address
Word access	Multiple of 2
Long-word access	Multiple of 4
Two long-word access	Multiple of 8

2.3.3.7 Addressing Modes Summary

Table 2-20 provides a summary of the addressing modes described in the previous sections. The Operand Reference columns are abbreviated as follows:

- S = Software Stack Reference in data memory (uses NSP or ESP according to mode)
- C = Program Control Unit Register Reference
- D = DALU Register Reference
- A = AGU Register Reference
- P = Program Memory Reference
- X = Data Memory Reference

Table 2-20. Addressing Modes Summary

Addressing Modes	R0-R7 Uses MCTL	Operand Reference						Assembler Syntax
		S	C	D	A	P	X	
Register Direct								
Data or Control Register	—		√	√				Dn Dn Dm Dn Dm Di Dj MCTL SR, EMR, VBA LC0, LC1 LC2, LC3 SA0, SA1 SA2, SA3
Address Register (Rn)	—				√			Rn
Address Modifier Register (Mj)	—				√			Mj
Base Address Register (Bn)	—				√			Bn
Address Offset Register (Ni)	—				√			Ni
Stack Pointer	—				√			SP



Table 2-20. Addressing Modes Summary (Continued)

Addressing Modes	R0-R7 Uses MCTL	Operand Reference						Assembler Syntax
		S	C	D	A	P	X	
Address Register Indirect								
No Update, (Rn)	No						√	(Rn)
Post-increment, (Rn)+	Yes						√	(Rn)+
Post-decrement, (Rn)-	Yes						√	(Rn)-
Post-increment by Offset Ni, (Rn)+Ni	Yes						√	(Rn) + Ni
Indexed by offset N0, (Rn+N0)	Yes						√	(Rn + N0)
Indexed by Address Register Rm, (Rn+Rm)	Yes						√	(Rn + Rm)
Short Displacement, (Rn+x) Word Displacement, (Rn+xxxx)	Yes						√	(Rn + x) (Rn + xxxx)
SP Short Displacement, (SP-xx)	—	√					√	(SP - xx)
SP Word Displacement, (SP+xxxx)	—	√					√	(SP + xxxx)
PC Relative								
PC Relative with Displacement	—						√	#xx (8 bits) #xxx (10 bits) #xxxx (16 bits) #xxxxx (20 bits)
Special								
Immediate Short Data Immediate Word Data Immediate Long Data	—						√	#xx (5, 6, or 7bits) #xxxx (16 bits) #xxxxxxxx(32 bits)
Absolute Word Address Absolute Long Address	—						√	xxxx (16 bits) xxxxxxxx (32 bits)
Absolute Jump Address	—						√	xxxxxxxx (32 bits)
Implicit Reference	—	√	√				√	

Note: The “—” that appears in the “R0-R7 Uses MCTL” heading means that it is not applicable for that addressing mode.

2.3.4 Address Modifier Modes

The AAU supports linear, reverse-carry, modulo, and multiple wrap-around modulo arithmetic types for address register indirect modes operating on R0-R7. These arithmetic types allow the easy creation of data structures in memory for First-In/First-Out (FIFO) queues, delay lines, circular buffers, stacks, and reverse-carry Fast Fourier Transform (FFT) buffers.

Data is manipulated by updating address registers (Rn) used as pointers rather than moving large blocks of data. The contents of the modifier control register MCTL define the type of arithmetic to be performed for address calculations. For modulo arithmetic, the address modifier register Mj specifies the modulus. Each of the address register lower banks (R0–R7) can be used with any of the modifier registers (M0–M3) as programmed in the MCTL register.

2.3.4.1 Linear Addressing Mode

Linear addressing is useful for general-purpose addressing such as stacks. In linear addressing mode, the address is calculated using standard binary arithmetic. The entire memory space is addressable. Linear addressing mode is selected by setting the AM3–0 bits to 0000 in the MCTL register. This is the default state.

2.3.4.2 Reverse-carry Addressing Mode

Reverse-carry addressing is useful for 2^k point FFT addressing. This mode is selected for R0-R7 by setting the AM3-0 bits to 0001 in the MCTL register. Address modification is performed in the hardware by propagating the carry from each pair of added bits in the reverse direction (from the MSB end toward the LSB end). For the $+N_i$ addressing mode, reverse-carry is equivalent to:

- Bit-reversing the contents of Rn (redefining the MSB as the LSB, the next MSB as bit 1, and so on)
- Shifting the offset value in N_i left by 0, 1, 2, or 3 according to the access width
- Bit-reversing the shifted N_i
- Adding normally
- Bit-reversing the result

This address modification is useful for addressing the twiddle factors in 2^k point FFT addressing as well as to unscramble 2^k point FFT data. The range of values for N_i is 0 to $2^{32}-1$, which allows reverse-carry addressing for FFTs up to 4,294,967,296 points.

Note: To achieve correct reverse-carry accessing for access widths of 2, 4, or 8, the last 1, 2, or 3 least significant bits (respectively) of the address calculation result are forced to zero.

2.3.4.3 Modulo Addressing Mode

Modulo address modification is useful for creating circular buffers for FIFO queues, delay lines, and sample buffers up to 2^{31} bytes long.

Modulo addressing is selected by writing the MCTL AM3-0 bits of the MCTL register (as shown in Table 2-10) as well as writing the desired modulus to the corresponding Mj register. Address modification is performed in modulo M, where M ranges from 1 to $+2^{31}-1$. Modulo M arithmetic causes the address register values to remain within an address range of size M, thus defining a buffer with a lower and an upper address boundary.

Each base address register (Bn register) is associated with an Rn register (B0 with R0, and so on). Each

register R_n has one M_j register assigned to it by encoding in the MCTL. The lower boundary value of the buffer resides in the B_n register, and the upper boundary is calculated as $B_n + M_j - 1$. M_j must be smaller than $2^{31} - 1$ ($M_j < 2^{31} - 1$).

The modulo addressing definition, using a base register (B_n) and a modulo register (M_j), enables the programmer to locate the modulo buffer at any address. The buffer start address is only required to be aligned to the access width.

The address pointer R_n is not required to start at the lower address boundary, nor to end on the upper address boundary. R_n can initially point anywhere (aligned to its access width) within the defined modulo address range, $B_n \leq R_n < B_n + M_j$. Assuming the $(R_n)^+$ indirect addressing mode, if the address register pointer increments past the upper boundary of the buffer (base address + $M_j - 1$), it wraps around through the base address (lower boundary). Alternatively, assuming the $(R_n)^-$ indirect addressing mode, if the address decrements past the lower boundary (base address), it wraps around through the base address + $M_j - 1$ (upper boundary).

The following constraints apply:

1. For proper modulo addressing, if an offset N_i is used in the address calculation, the 32-bit absolute effective value $|N_i|$ must be less than or equal to M_j , where “effective” means the programmed N_i is multiplied by the access width. For example, `move.w (r0)+n0,d0` translates to the restriction $2 * n0 \leq M_j$, and `move.l (r0)+,d0` translates to $4 \leq M_j$. If effective $N_i > M_j$, the result of the address calculation is undefined. Multiple wrap-around modulo addressing supports the situation of effective N_i greater than M_j .
2. M_j must be aligned to the access width used. For example, if the buffer is used with a `MOVE.2L` instruction, M_j must be aligned to 8 (be a multiple of 8). If the modulus is less than the access width, the data accessed as well as the address calculations are undefined.
3. When B_n is used as a base address register, the use of R_{n+8} as a pointer is illegal since this is the same physical register.

Modulo addressing is illustrated in Figure 2-15. Addresses will be kept within the eleven addresses shown. For the instruction, `move.w (r0+$000e),d0`, the access will be made from \$26 (38), if the base address is \$20, the modulus is \$c, and $r0$ is \$24. The operation is $36 + 14 = 50 = 38$ in modulus 12, base address 32 ($50 - 44 + 32 = 38$).

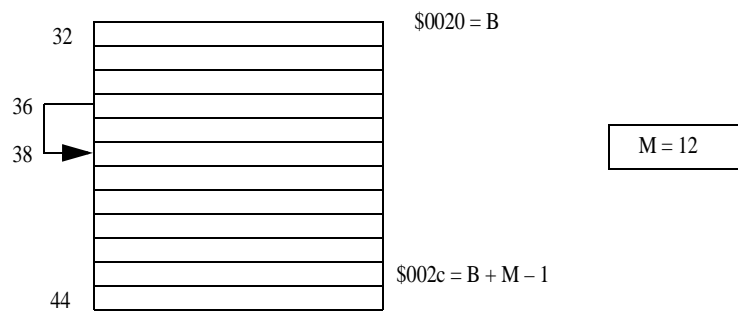


Figure 2-15. Modulo Addressing Example

Table 2-21 describes the modulo register values and the corresponding address calculation.

Table 2-21. Modulo Register Values for Modulo Addressing Mode

Modifier Mj	Address Calculation Arithmetic
\$0000 0000	Unused
\$0000 0001	Modulo 1
\$0000 0002	Modulo 2
\$7FFF FFFE	Modulo $2^{31}-2$
\$7FFF FFFF	Modulo $2^{31}-1$

2.3.4.4 Multiple Wrap-Around Modulo Addressing Mode

Multiple wrap-around addressing is useful for decimation, interpolation, and waveform generation. The multiple wrap-around capability can be used for argument reduction. In multiple wrap-around modulo addressing mode, the modulus M is a power of 2 in the range of 2^1 to 2^{31} . The value $M-1$ is stored in the modifier register (M_j). The B registers B0 to B7 are not used for multiple wrap-around modulo addressing; therefore, their corresponding R8–R15 registers can be used for linear addressing.

The lower and upper boundaries are derived from the contents of M_j . The lower boundary (base address) value has zeros in the k LSBs where $M = 2^k$ and therefore must be a multiple of M . The R_n register involved in the memory access is used to set the MSBs of the base address. The base address is set so that the initial value in the R_n register is within the lower and upper boundaries. The upper boundary is the lower boundary plus the modulo size minus one (base address + $M-1$).

The size of the modulo buffer must be aligned to (be a multiple of) the access width. If the modulus is less than the access width, the data accessed as well as the address calculations are undefined.

If an offset N_i is used in the address calculations, it is not required to be less than or equal to M for proper modulo addressing. The multiple wrap-around modulo addressing mode supports unlimited boundary wraps.

When using the $(R_n)_+$ and $(R_n)_-$ addressing modes with a modulus $2^k \geq 8$, there is no functional difference between the multiple wrap-around and normal modulo modes since the address can only be wrapped around once.

As an example, consider the instruction `move.w (r0 + $0042),d0`. If the `mctl` is set to `$000c`, and `m0` is set to `$000f`, then $M_0 = 16$. If `r0` is initially `$24 (36)`, the lower boundary is `$20 (32)` and the upper boundary is `$2f (47)`. The memory access is done from address `$26 (38)`, calculated by $36 + 66 = 102$, $102 - 48 = 54$, $54 - 3 \times 16 = 6$, $6 + 32 = 38$.

Table 2-22 describes the modulo register Mj values and the corresponding multiple wrap-around address calculation.

Table 2-22. Modulo Register Values for Wrap-Around Modulo Addressing Mode

Modifier Mj	Address Calculation Arithmetic
\$0000 0001	Multiple Wrap-around Modulo 2
\$0000 0003	Multiple Wrap-around Modulo 4
\$0000 0007	Multiple Wrap-around Modulo 8
\$7FFF FFFF	Multiple Wrap-around Modulo 2^{31}
\$FFFF FFFF	Linear

2.3.5 Arithmetic Instructions on Address Registers

The SC140 core provides arithmetic instructions on the address registers (R0–R15), offset registers (N0–N3), the stack pointer (SP), and the program counter (PC).

Address modification modes can affect the arithmetic results stored in R0-R7 using instructions ADDA, SUBA, ADDL1A, or ADDL2A. In addition, an address calculation that increments or decrements address register R0-R7 is affected by the modifier mode. When updating R0-R7 in modulo addressing mode, the modulo registers hold the modulus.

Table 2-23 lists the arithmetic instructions that are executed in the AGU unit. A more detailed description of the operations is provided in [Appendix A, “SC140 DSP Core Instruction Set.”](#)

Table 2-23. AGU Arithmetic Instructions

Instruction	Description
ADDA	AGU Add (affected by the modifier mode)
ADDL2A	AGU Add with 2-bit left shift of source operand (affected by the modifier mode)
ADDL1A	AGU Add with 1-bit left shift of source operand (affected by the modifier mode)
ASL2A	AGU Arithmetic shift left by 2 bits (32-bit)
ASLA	AGU Arithmetic shift left (32-bit)
ASRA	AGU Arithmetic shift right (32-bit)
CMPEQA	AGU Compare for equal
CMPGTA	AGU Compare for greater than
CMPHIA	AGU Compare for higher (unsigned)
DECA	AGU Decrement register (affected by the modifier mode)
DECEQA	AGU Decrement and set T if result is zero

Table 2-23. AGU Arithmetic Instructions (Continued)

Instruction	Description
DECGEA	AGU Decrement and set T if result is equal to or greater than zero
INCA	AGU Increment register (affected by the modifier mode)
LSRA	AGU Logical shift right (32-bit)
SUBA	AGU Subtract (affected by the modifier mode)
SXTA.B	AGU Sign-extend byte
SXTA.W	AGU Sign-extend word
TFRA	AGU Register transfer
TSTEQA	AGU Test for equal to zero
TSTEQA.W	AGU Test for equal to zero on lower 16 bits
TSTGEA	AGU Test for greater than or equal to zero
TSTGTA	AGU Test for greater than zero
ZXTA.B	AGU Zero-extend byte
ZXTA.W	AGU Zero-extend word

2.3.6 Bit Mask Instructions

The SC140 core provides bit mask instructions on all address registers (R0–R15), all DALU registers (D0–D15), all control registers (EMR, VBA, SR, MCTL), and all memory locations.

Bit mask instructions provide an easy way of setting, clearing, inverting, or testing a selected but not necessarily adjacent group of bits in a register or memory location.

All bit mask instructions work on 16-bit data. This data can be the contents of a memory location or a portion (high or low) of a register.

Only a single bit mask instruction is allowed in one execution set since only one execution unit exists for these instructions. A subgroup of the bit mask instructions (BMTSET) supports hardware semaphores. For more information, see [Section 2.3.6.1, “Bit Mask Test and Set \(Semaphore Support\) Instruction.”](#)



Table 2-24 lists the arithmetic instructions that are executed in the BMU.

Table 2-24. AGU Bit Mask Instructions (BMU)

Instruction	Description
AND.W	Logical AND on a 16-bit operand
BMCHG	Bit mask change Inverts every bit in the destination (register or memory) that has the value 1 in the mask.
BMCLR	Bit mask clear Clears every bit in the destination (register or memory) that has the value 1 in the mask.
BMSET	Bit mask set Sets every bit position in the destination (register or memory) that has the value 1 in the mask.
BMTSET	Bit mask test (if set) and set Sets the T bit if every bit that has the value 1 in the mask is 1 in the destination (register or memory). Sets (writes) every bit in the destination (register or memory) that has the value 1 in the mask, and sets the T-bit if the set (write) failed. See Section 2.3.6.1, “Bit Mask Test and Set (Semaphore Support) Instruction.”
BMTSTC	Bit mask test if clear Sets the T-bit, if every bit position that has the value 1 in the mask is 0 in an operand.
BMTSTS	Bit mask test if set Sets the T bit if every bit position that has the value 1 in the mask is 1 in an operand.
EOR.W	Logical exclusive OR on a 16-bit operand
NOT.W	Binary inversion of a 16-bit operand
OR.W	Logical OR on a 16-bit operand

2.3.6.1 Bit Mask Test and Set (Semaphore Support) Instruction

The bit mask test and set instruction (BMTSET) provides support for hardware semaphores. A semaphore is a signal which can be set to indicate whether a program resource can be accessed or not. The destination of this instruction can be a register or a memory location in either internal or external memory. If the semaphore indicates that the resource is available, the T bit has the value 0. If the semaphore indicates that the resource is not available ($T = 1$), a jump can be made to skip the resource code.

This instruction performs the following tasks:

1. Reads the destination register, tests the data, and sets the T bit, if every bit that has the value 1 in the mask is 1 in the destination.
2. Writes back to the destination a word with ones for the masked bits, and the original destination bits for the unmasked bits.
3. Sets the T bit if the set (write) failed.

Normally, the BMTSET consists of three indivisible operations: read, update the T bit, and write. A set (write) failed condition occurs if the destination failed to be written indivisibly from the previous read operation of that BMTSET instruction. The memory subsystem signals the core of a write failure if a memory access that is initiated by another master source intervenes between the read and the write accesses of the BMTSET operation. As a result of the non-exclusive write indication, the T bit is set, signalling that the resource may not be available, thereby avoiding a hazard condition.

2.3.6.1.1 Example of Normal Usage of the Semaphoring Mechanism

The following sequence accesses a resource controlled by a semaphore.

```
label : BMTSET.W #mask, (R0)
JT label
```

Normally, the mask enables only one bit. In this case, the memory destination pointed to by (R0) is read, and the enabled bit is tested. The enabled bit is then set, and the memory destination is written back.

The T bit is set if the enabled bit was originally 1 (meaning that it was semaphore-occupied), or that the write-back failed. A T bit value of TRUE indicates to the conditional jump that the attempt to obtain the resource has failed, and that the jump should be taken. The T bit is cleared if the enabled bit was originally zero. This means that the semaphore was not allocated. Therefore, the resource was available, and the instruction was successful in setting the semaphore exclusively. A successful allocation writeback results.

When the destination is a register, the write is always successful.

2.3.6.2 Semaphore Hardware Implementation

During the address phase of the read and write accesses associated with the BMTSET instruction, an output of the core is asserted. This assertion indicates that the read and the following write are part of a read-modify-write sequence.

During the data phase of the write access, a core input provides the core with the result of the access (de-asserted = write failed).

2.3.7 Move Instructions

The SC140 instruction set supports various types of move instructions which differ in the following properties:

- Access width — Byte (8 bits), word (16 bits), long-word (32 bits), and two long words (64 bits)
- Data type — Signed integer, unsigned integer, fractional (with or without limiting)
- Multi-register moves — Some move operations split data between two or four registers
- Addressing mode — For example, absolute, relative to an address pointer (with various offset and post-update options), and relative to the stack pointer

The move instructions perform data movement over the XDBA and XDBB buses (for data moves). Move instructions do not affect the status register with the exception of the sticky scaling bit in reading a DALU register.

Table 2-25 lists the move instructions. The suffix just before the period in the MOVE nomenclature indicates the following:

- None = Signed
- U = Unsigned
- S = Scaling and limiting (saturation) enabled

The suffix just after the period in the MOVE nomenclature indicates the following:

- B = Byte
- W = Integer word (16 bits)
- L = Long word (32 bits)
- F = Fractional word (16 bits)

Either a two or four may modify the last suffix.

Table 2-25. AGU Move Instructions

Instruction	Description
MOVE.2F	Move two fractional words from memory to a register pair
MOVE.2L	Move two longs to/from a register pair
MOVE.2W	Move two integer words to/from memory and a register pair
MOVE.4F	Move four fractional words from memory to a register quad
MOVE.4W	Move four integer words to/from memory and a register quad
MOVE.B	Move byte to/from memory
MOVE.F	Move fractional word to/from memory
MOVE.L	Move long
MOVE.W	Move integer word to/from memory, or immediate to register or memory
MOVEc	Conditional move between address registers
MOVES.2F	Move two fractional words to memory with scaling and limiting enabled
MOVES.4F	Move four fractional words to memory with scaling and limiting enabled
MOVES.F	Move fractional word to memory with scaling and limiting enabled
MOVES.L	Move long to memory with scaling and limiting enabled
MOVEU.B	Move unsigned byte from memory
MOVEU.L	Move unsigned long from immediate
MOVEU.W	Move unsigned integer word from memory or from immediate
VSL.2F	Viterbi shift left—specialized move to support Viterbi kernel
VSL.2W	Viterbi shift left—specialized move to support Viterbi kernel
VSL.4F	Viterbi shift left—specialized move to support Viterbi kernel
VSL.4W	Viterbi shift left—specialized move to support Viterbi kernel

Integer moves from memory (byte, word, long, two long) are right-aligned in the destination register, and by default are sign-extended to the left. Unsigned moves are marked with “U” (for example, MOVEU.B), and zero extended in the destination register. A schematic representation of integer moves from memory into a 40-bit register is shown in Figure 2-16. Moves from registers to memory use the appropriate portion from the source register. Moves to registers of less than 40 bits behave the same as in Figure 2-16 up to their bit length.

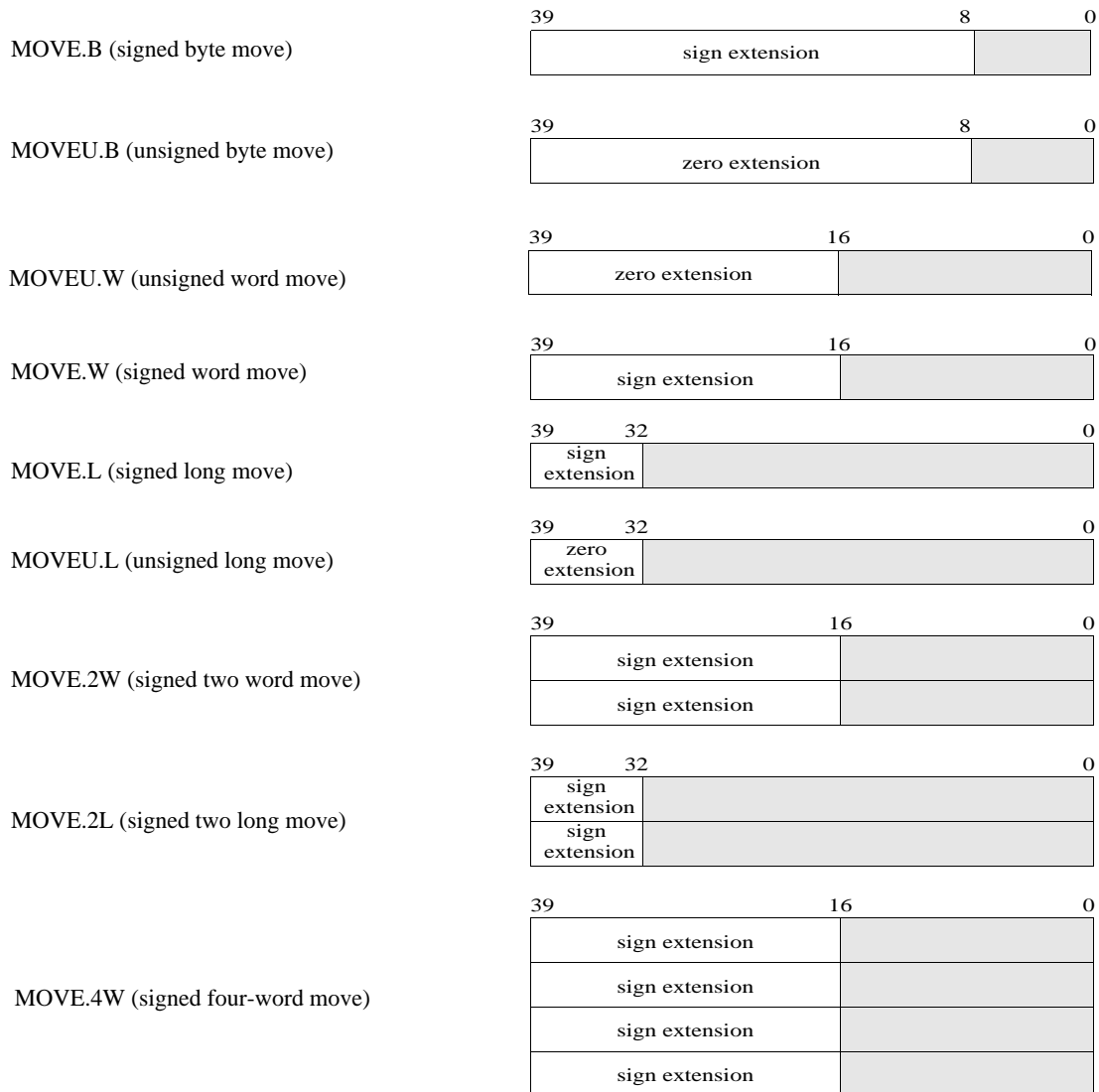


Figure 2-16. Integer Move Instructions

Fractional moves are supported only to DALU registers. Moves from memory are put in the high portion of the data register, sign-extended to the extension, and zero-filled in the low portion. MOVE.L and MOVE.2L may also be considered fractional moves since alignment in the destination register is the same for integer long moves and fractional long moves. A schematic representation of fractional moves from memory to 40-bit data registers is shown in Figure 2-17.

The extension bits of the even data register occupy bits 0 to 8 (bit 8 is the limit bit). The extension bits of the odd register occupy bits 16 to 24 (bit 24 is the limit bit) as described in Figure 2-18.

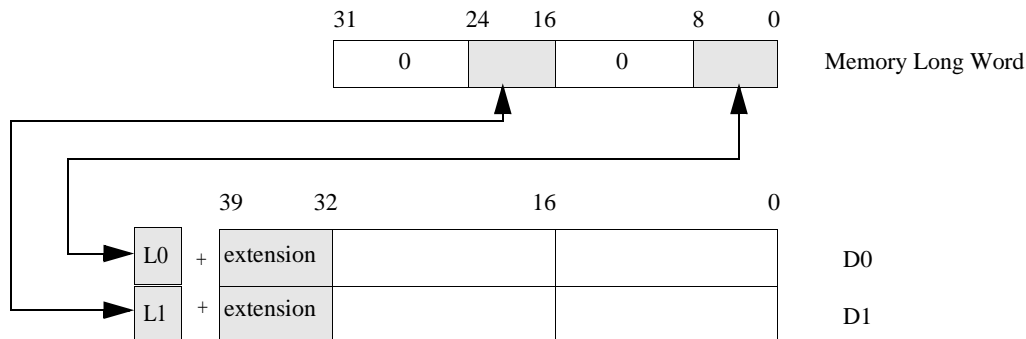


Figure 2-18. Bit Allocation in MOVE.L D0.e:D1.e

Moves from memory to an extension are only to single registers. However, they are also 32-bit wide and implicitly assume the bit allocation described above according to the register number (odd or even). For example, `move.l $4F42,d3.E` is the instruction for moving bits 24:16 from the memory location addressed by \$4F42 to the limit bit and extension bits of the odd register d3. See [Appendix A, “Move Long Word \(AGU\) MOVE.L,”](#) for more information about the moves to and from extension registers.

2.4 Memory Interface

The SC140 core interfaces to memory via the following:

- 32-bit program memory address bus (PAB) and 128-bit program memory data bus (PDB)
- 32-bit data memory address bus A (XABA) and 64-bit data memory data bus A (XDBA)
- 32-bit data memory address bus B (XABB) and 64-bit data memory data bus B (XDBB)
- Control signals such as read and write access strobes as well as access width control

The SC140 does not specify a memory subsystem architecture, only the minimum requirements for correct execution of SC140 code. Listed below are requirements for all memory designs that interface with the SC140 core.

- The SC140 core supports only unified memory designs. Memory is regarded as a single space. There is no distinction between program memory locations and data memory locations. Each memory location possesses a unique address that can be accessible from either the program or data buses. From the core’s perspective, there is only one memory address “a,” which can hold either data or program information.
- Data must be byte-addressable and accessible by the two data memory buses.
- All data width accesses used by the SC140 core must be supported by the memory such as byte (8 bits), word (16 bits), long word (32 bits), or double-long word and four-word (64 bits). One of four control signals will indicate to the memory which access width is needed for each access.
- Multi-byte memory accesses must support both endian modes.

- Memory must resolve access ordering on a cycle by cycle basis. All accesses on a given cycle must be completed before proceeding to accesses in the next cycle. Note that a conflict access may occur when there are multiple requests to access the same memory module, in the same cycle. An access conflict is resolved by a stall cycle (per conflict), which serializes the multiple request.
- Multiple access rules in a given cycle are as follows:
 - Multiple read or write accesses to different memory locations execute without any predetermined sequence.
 - In cases where multiple accesses to the same memory location occur, the access sequence is program fetch, data read, and data write.
 - If two write operations access the same byte in memory in the same cycle, the operation is illegal and the result is undefined. The same byte may be written by different but overlapping words or long words. The memory subsystem should be able to detect these cases and issue an imprecise interrupt to the core. The use of this interrupt is optional. Refer to [Section 5.3.3.2, “Implicit Push/Pop Memory Timing,”](#) on page 5-24 for more details.
- Accesses to non-existent memory locations are illegal and the result is undefined. The memory subsystem can issue an imprecise interrupt to the core. The use of this interrupt is optional.

2.4.1 SC140 Endian Support

The term “little endian” is defined as a computer architecture such that given a multi-byte operand representation, bytes at lower addresses have lower numeric significance. Each word is stored little end first. In little endian mode, the `MOVE.W D0,(R0)` instruction (for example) stores bits 7–0 of D0 into address (R0), and bits 15–8 into address (R0 + 1).

In “big endian” architectures, the most significant byte has the lowest address, and each word is stored big end first. In big endian mode, the `MOVE.W D0,(R0)` instruction stores bits 15–8 of D0 into address (R0), and bits 7–0 into address (R0 + 1).

The SC140 supports both big and little endian architectures through the big endian memory (BEM) mode bit in the EMR. This bit samples a core input signal when exiting the reset state, and cannot be changed during normal operation.

Figure 2-19 shows an example how data is transferred from a register to memory in the two endian modes.

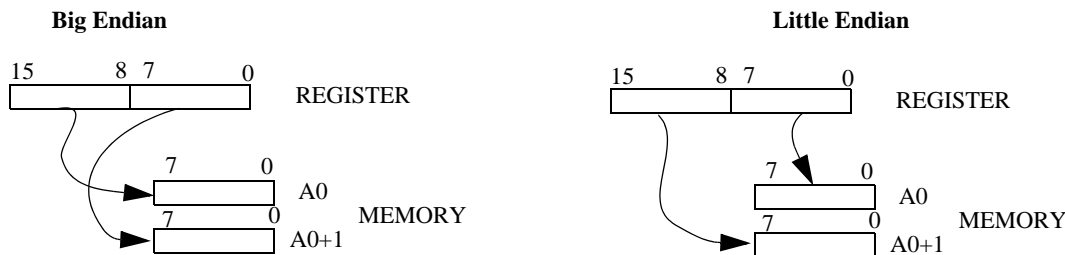


Figure 2-19. Endian Example

2.4.1.1 SC140 Bus Structure

The entire memory space of the SC140 core is unified. The memory supports two parallel 64-bit data accesses and one 128-bit program fetch. All can occur in parallel.

The two data buses that connect between the core and the memory are each 64 bits wide. Instructions such as load to registers and store to memory utilize the bus according to the application requirement. Different versions of the instructions are used for different bandwidths such that:

- MOVE.B loads or stores bytes (8 bits).
- MOVE.W and MOVE.F load or store integer or fractional words (16 bits).
- MOVE.2W, MOVE.2F, and MOVE.L load or store double-integers, double-fractions, and long words respectively (32 bits).
- MOVE.4W, MOVE.4F, and MOVE.2L load or store quad-integers, quad-fractions, and double-long words respectively (64 bits).

Figure 2-20 shows the data busses between the SC140 core and the memory.

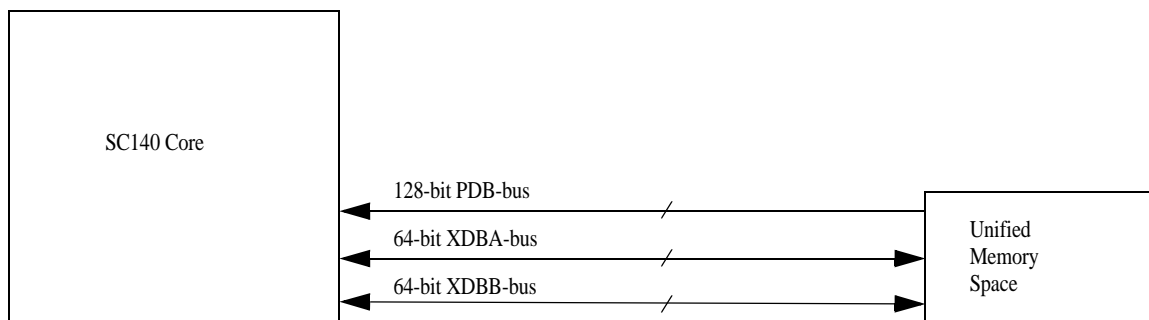


Figure 2-20. Basic Connection between SC140 Core and Memory

2.4.1.2 Memory Organization

Different types of data are stored differently in memory for each of the two endian modes. However, the data retains the same meaning. For example, 64 bits of data can be represented by any of the following:

- Eight 8-bit bytes
- Four 16-bit numbers
- Two 32-bit numbers

Figure 2-21 shows how data is organized in memory in the two endian modes. Each data unit is a byte made of two hexadecimal numbers.



Figure 2-21. Memory Organization of Big and Little Endian Mode

Table 2-26 describes the data representation for each 64-bit row in Figure 2-21.

Table 2-26. Data Representation in Memory

Representation Type	Value
Eight 8-bit bytes	A0 = \$0a, A1 = \$0b, A2 = \$0c, A3 = \$0d, A4 = \$0e, A5 = \$0f
Four 16-bit numbers	A8 = \$0102, A10 = \$0304, A12 = \$0506, A14 = \$0708
Two 32-bit numbers	A16 = \$11223344, A20 = \$ccddeeff

2.4.1.3 Data Moves

Data moves are executed by moving core registers to and from memory over one of the data buses (XDBA or XDDB).

The data registers can be accessed with three types of data:

- Long type access, writing or reading 32-bit operands.
- Word type access, writing or reading 16-bit operands.
- Byte type access, writing or reading 8-bit operands.

Figure 2-22 shows an example of data transfer in big and little endian modes.

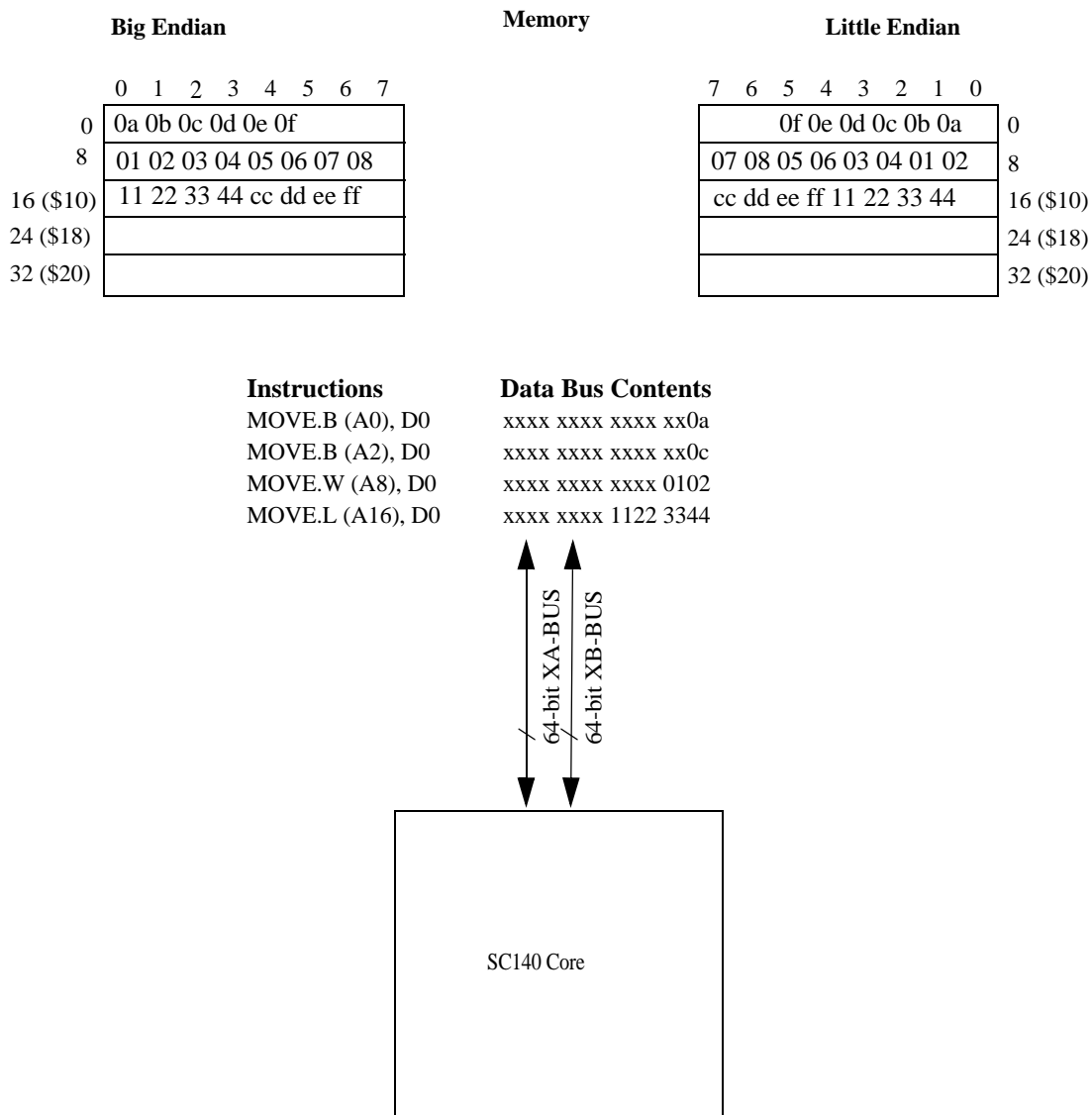


Figure 2-22. Data Transfer in Big and Little Endian Modes

For single-register moves, assuming an equivalent memory map in big and little endian modes, the byte organization on the buses is identical in both modes. However, the memory subsystem must route the data bus bytes to different memory addresses for each supported endian mode.

2.4.1.4 Multi-Register Moves

For accesses involving more than one register, such as with `MOVE.2W` or `MOVE.4F` instructions, the SC140 ensures that data originating from a specific register reaches the same address in memory in both little and big endian modes (and the other way round). The memory system does not distinguish between `MOVE.L` and `MOVE.2W` transfers that have the same data width. Memory treats them both like a long word transfer. If the data bus were the same for both endian modes in a two-register transfer, the data from the two registers would end up in different addresses. To correct for this, the byte order on the buses for multi-register transfers is adjusted for the little endian mode. The memory also does not distinguish between transfers of four words or two long words. It treats them both like a string of eight bytes. The bus structure for the little endian mode corrects for both cases to ensure that register data is stored at the same address for both modes.

As an example of the problem that arises if a correction is not made, consider the following case:

The instruction `move.l 2w d0:d1, (a8)` transfers two integer words from data registers `d0` and `d1` to memory at address `a8`. For `d0 = $0102` and `d1 = $0304`, the data bus would be `$01020304`, and the memory would be accessed for a width of 32 bits. For big endian mode, the memory would look like:

Address	Data
a8	01
a9	02
a10	03
a11	04

For little endian mode, the memory would be accessed for a width of 32 bits (like a long word), and then it would write the data little end first such that the memory would look like:

Address	Data
a8	04
a9	03
a10	02
a11	01

Note that the data word from `d0`, `$0102`, is at a different address for the two modes. If the data bus were modified by the core to `$03040102`, then the memory for little endian mode would look like:

Address	Data
a8	02
a9	01
a10	04
a11	03

This is the desired result. This effect is achieved in little endian mode through logic in the core, which modifies the data on the data bus to the memory for both reads and writes.

Figure 2-23 shows examples of multi-register data transfers in big and little endian modes.

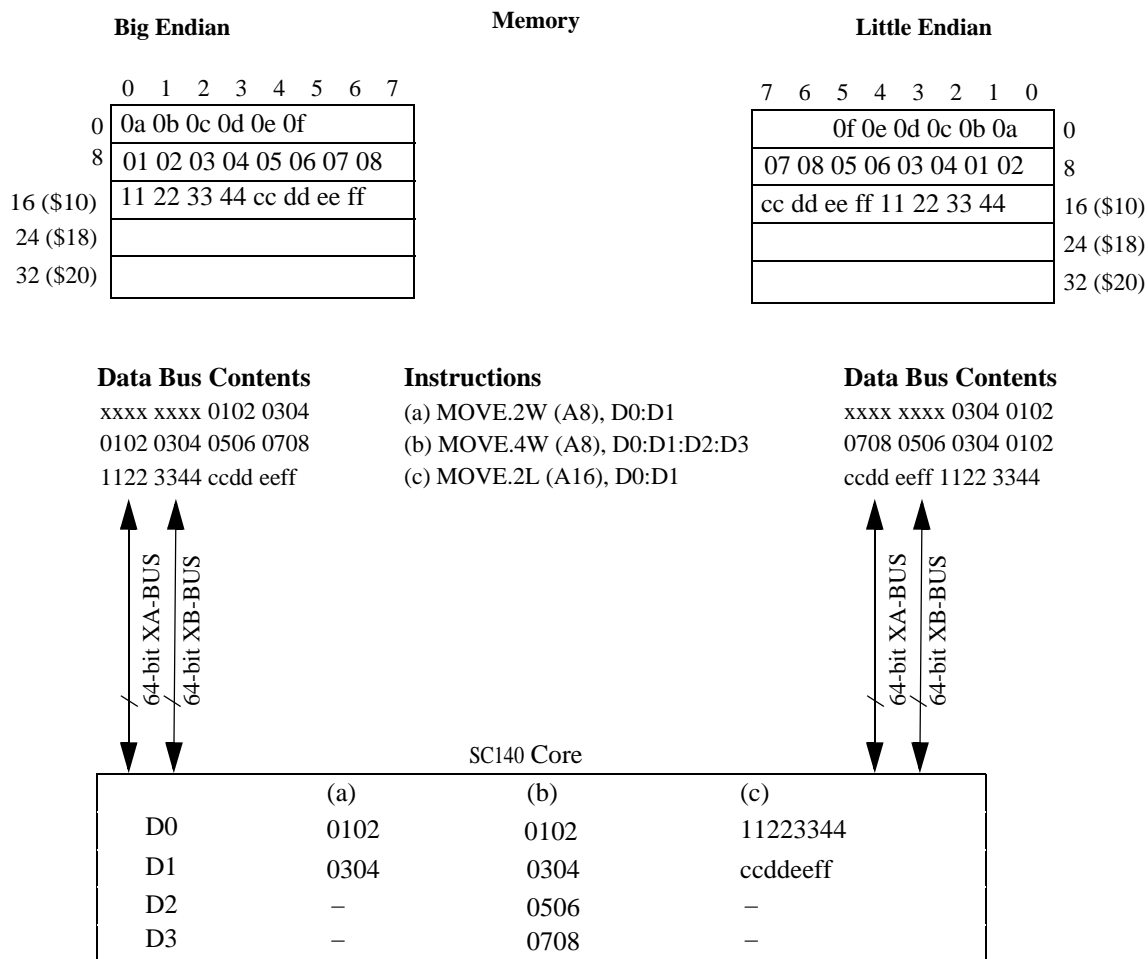


Figure 2-23. Multi-Register Transfer in Big and Little Endian Modes

Note: The only exceptions to the behavior described above are the VSL instructions. These instructions cause source data words from the core to be written to different memory locations in big and little endian modes. For more information about the VSL instructions, refer to Table 2-27 on page 2-64, and [Appendix A, “Viterbi Shift Left Move \(AGU\) VSL,”](#) on page A-422..

2.4.1.5 Instruction Word Transfers

Instruction words are transferred to the core from memory over the program data bus (PDB) to special instruction registers in the program dispatch unit (PDU).

The instruction registers can be accessed only with aligned access of 128-bit width (8 instruction words). Figure 2-24 shows the program memory organization in big and little endian modes. Note that program data consists of a series of 16-bit instructions. In this example the assembler determines the instructions to be:

```
word address $00 instruction $a0b0
word address $02 instruction $c0d0
word address $04 instruction $e0f0
word address $06 instruction $a1b1
word address $08 instruction $c1d1
word address $0a instruction $e1f1
word address $0c instruction $a2b2
word address $0e instruction $c2d2
word address $10 instruction $e2f2
.....
```

These are to be placed in memory as shown in the following figure.

Big Endian				Little Endian				
0	1	2	3	4	5	6	7	
a0b0	c0d0	e0f0	a1b1	a1b1	e0f0	c0d0	a0b0	0
c1d1	e1f1	a2b2	c2d2	c2d2	a2b2	e1f1	c1d1	8
e2f2	a3b3	c3d3	e3f3	e3f3	c3d3	a3b3	e2f2	16 (\$10)
								24 (\$18)

Figure 2-24. Program Memory Organization in Big and Little Endian Modes

The assembler outputs a byte stream to the loader and therefore corrects for the byte address reversal inside each 16-bit instruction to achieve the memory results above.

```
Big Endian Assembler Output Little Endian Assembler Output
byte address $00 data $a0byte address $00 data $b0
byte address $01 data $b0byte address $01 data $a0
byte address $02 data $c0byte address $02 data $d0
byte address $03 data $d0byte address $03 data $c0
byte address $04 data $e0byte address $04 data $f0
.....      .....
```

Figure 2-25 shows the memory accesses to the same memory area by both program fetches as well as data accesses in big and little endian modes.

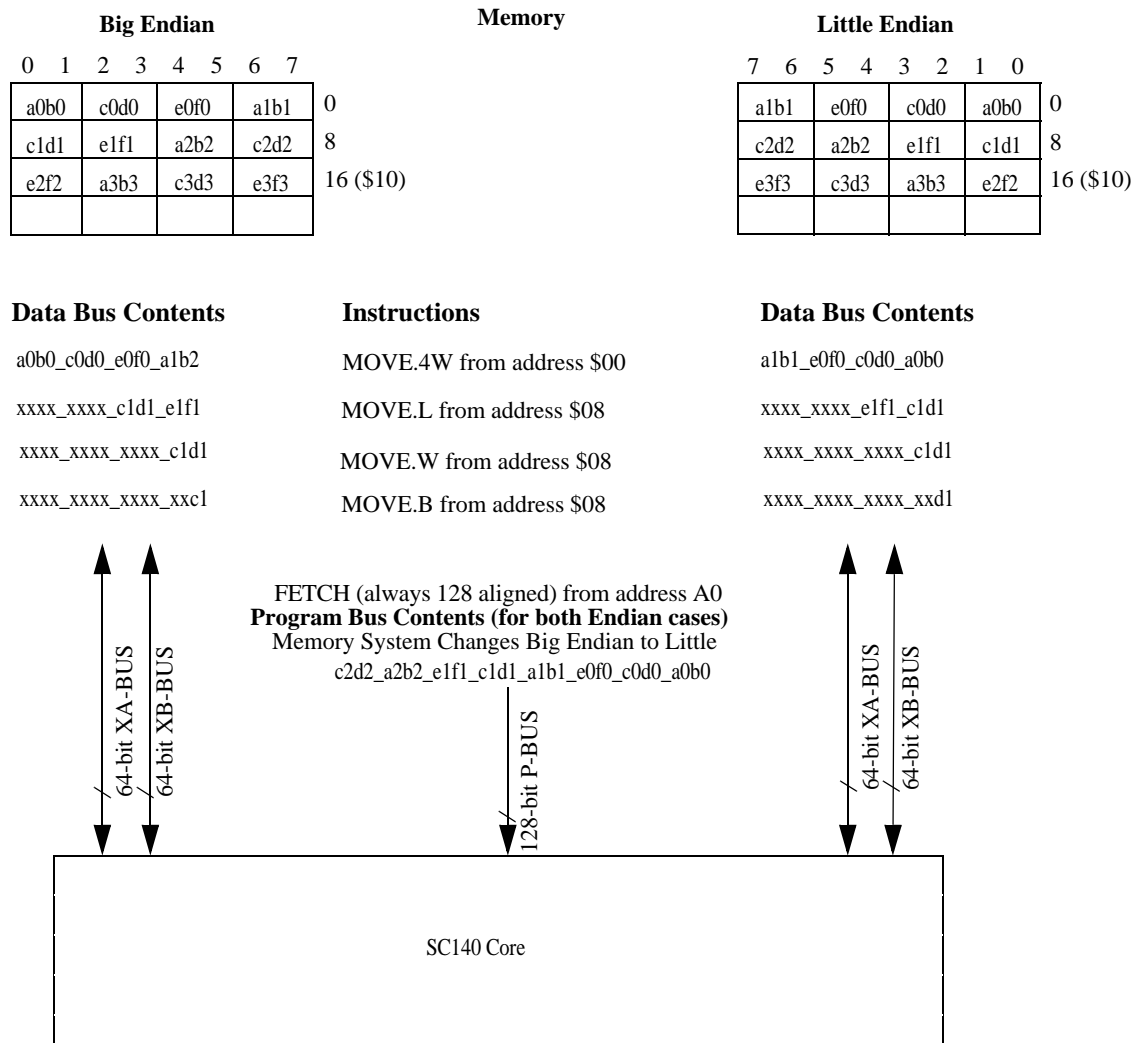


Figure 2-25. Instruction Moves in Big and Little Endian Modes

The Program Bus contents always appear as eight 16-bit little endian packed instructions, the memory system performing a word (instruction) reversal in the case of big endian (program bus only).

2.4.1.6 Memory Access Behavior in Big/Little Endian Modes

Table 2-27 shows the representation of the move instructions in big and little endian modes. In the examples shown in this table, it is assumed that R0 points to address A0. Each alphanumeric A–H represents one byte. Also, the memory contents may not exactly equal the register contents. For example, in VSL instructions, the memory word (16 bits) is the register word shifted left by one bit. See Appendix A for more detailed information.

Table 2-27. Move Instructions in Big and Little Endian Modes

Instruction	Register Operands	Big Endian	Little Endian												
MOVE.B MOVEU.B	Example: MOVE.B D0,(R0) 39 8 0 D0 = <table border="1" style="display: inline-table; vertical-align: middle;"><tr><td style="width: 24px;"></td><td style="width: 8px; text-align: center;">A</td></tr></table>		A	A0 = A	A0 = A										
	A														
MOVE.W MOVEU.W	Example: MOVE.W D0, (R0) 39 16 0 D0 = <table border="1" style="display: inline-table; vertical-align: middle;"><tr><td style="width: 16px;"></td><td style="width: 8px; text-align: center;">A</td><td style="width: 8px; text-align: center;">B</td></tr></table>		A	B	A0 = A A1 = B	A0 = B A1 = A									
	A	B													
MOVE.2W	Example: MOVE.2W D0:D1, (R0) 39 16 0 D0 = <table border="1" style="display: inline-table; vertical-align: middle;"><tr><td style="width: 16px;"></td><td style="width: 8px; text-align: center;">A</td><td style="width: 8px; text-align: center;">B</td></tr></table> D1 = <table border="1" style="display: inline-table; vertical-align: middle;"><tr><td style="width: 16px;"></td><td style="width: 8px; text-align: center;">C</td><td style="width: 8px; text-align: center;">D</td></tr></table>		A	B		C	D	A0 = A A1 = B A2 = C A3 = D	A0 = B A1 = A A2 = D A3 = C						
	A	B													
	C	D													
MOVE.4W	Example: MOVE.4W D0:D1:D2:D3, (R0) 39 16 0 D0 = <table border="1" style="display: inline-table; vertical-align: middle;"><tr><td style="width: 16px;"></td><td style="width: 8px; text-align: center;">A</td><td style="width: 8px; text-align: center;">B</td></tr></table> D1 = <table border="1" style="display: inline-table; vertical-align: middle;"><tr><td style="width: 16px;"></td><td style="width: 8px; text-align: center;">C</td><td style="width: 8px; text-align: center;">D</td></tr></table> D2 = <table border="1" style="display: inline-table; vertical-align: middle;"><tr><td style="width: 16px;"></td><td style="width: 8px; text-align: center;">E</td><td style="width: 8px; text-align: center;">F</td></tr></table> D3 = <table border="1" style="display: inline-table; vertical-align: middle;"><tr><td style="width: 16px;"></td><td style="width: 8px; text-align: center;">G</td><td style="width: 8px; text-align: center;">H</td></tr></table>		A	B		C	D		E	F		G	H	A0 = A A1 = B A2 = C A3 = D A4 = E A5 = F A6 = G A7 = H	A0 = B A1 = A A2 = D A3 = C A4 = F A5 = E A6 = H A7 = G
	A	B													
	C	D													
	E	F													
	G	H													
MOVE.L MOVEU.L MOVES.L	Example: MOVE.L D0, (R0) 39 32 0 D0 = <table border="1" style="display: inline-table; vertical-align: middle;"><tr><td style="width: 8px;"></td><td style="width: 8px; text-align: center;">A</td><td style="width: 8px; text-align: center;">B</td><td style="width: 8px; text-align: center;">C</td><td style="width: 8px; text-align: center;">D</td></tr></table>		A	B	C	D	A0 = A A1 = B A2 = C A3 = D	A0 = D A1 = C A2 = B A3 = A							
	A	B	C	D											

Table 2-27. Move Instructions in Big and Little Endian Modes (Continued)

Instruction	Register Operands	Big Endian	Little Endian
MOVE.L (Extension)	Example: MOVE.L D0.E:D1.E, (A0) 	A0 = L1 A1 = B A2 = L0 A3 = A	A0 = A A1 = L0 A2 = B A3 = L1
MOVE.2L	Example: MOVE.2L D0:D1, (R0) 	A0 = A A1 = B A2 = C A3 = D A4 = E A5 = F A6 = G A7 = H	A0 = D A1 = C A2 = B A3 = A A4 = H A5 = G A6 = F A7 = E
MOVE.F MOVES.F	Example: MOVE.F D0, (R0) 	A0 = A A1 = B	A0 = B A1 = A
MOVE.2F MOVES.2F	Example: MOVE.2F D0:D1, (R0) 	A0 = A A1 = B A2 = C A3 = D	A0 = B A1 = A A2 = D A3 = C
MOVE.4F MOVES.4F	Example: MOVE.4F D0:D1:D2:D3, (R0) 	A0 = A A1 = B A2 = C A3 = D A4 = E A5 = F A6 = G A7 = H	A0 = B A1 = A A2 = D A3 = C A4 = F A5 = E A6 = H A7 = G

Table 2-28 shows the representation of the stack support instructions in big and little endian modes. In the examples shown in this table, it is assumed that the stack access is to address A0. The stack instructions treat the register data like a 32-bit long word move.

Table 2-28. Stack Support Instructions in Big and Little Endian Modes

Instruction	Register Operands	Big Endian	Little Endian
Single POP POPN PUSH PUSHN	Example: PUSH D0 $D0 = \begin{array}{ c c c c } \hline & 31 & & 0 \\ \hline A & B & C & D \\ \hline \end{array}$	A0 = A A1 = B A2 = C A3 = D	A0 = D A1 = C A2 = B A3 = A
Double POP POPN PUSH PUSHN	Example: PUSH D0 PUSH D1 $D0 = \begin{array}{ c c c c } \hline & 31 & & 0 \\ \hline A & B & C & D \\ \hline \end{array}$ $D1 = \begin{array}{ c c c c } \hline E & F & G & H \\ \hline \end{array}$	A0 = A A1 = B A2 = C A3 = D A4 = E A5 = F A6 = G A7 = H	A0 = D A1 = C A2 = B A3 = A A4 = H A5 = G A6 = F A7 = E

Table 2-29 shows the representation of the bit mask instructions in big and little endian modes.

Table 2-29. Bit Mask Instructions in Big and Little Endian Modes

Instruction	Register Operands	Big Endian	Little Endian
BMCHG.W BMCLR.W BMSET.W BMTSTS.W BMTSTC.W BMTSET.W NOT.W AND.W OR.W EOR.W	Example: BMSET.W #1234, (A0) $Data = \begin{array}{ c c } \hline & 15 & & 0 \\ \hline A & B \\ \hline \end{array}$ $Mask = \quad 12 \quad \quad 34$	A0 = A A1 = B	A0 = B A1 = A

Table 2-30 shows the representation of the change-of-flow instructions in big and little endian modes. In this table, it is assumed that the stack access is to address A0. This shows how the contents of the PC and SR are transferred to/from memory like 32-bit long words.

Table 2-30. Non-Loop Change-of-Flow Instructions in Big and Little Endian Modes

Instruction	Register Operands	Big Endian	Little Endian
BSR BSRD JSR JSRD RTE RTED	$ \begin{array}{c} \text{PC} = \begin{array}{ c c c c } \hline & 31 & & 0 \\ \hline A & B & C & D \\ \hline \end{array} \\ \text{SR} = \begin{array}{ c c c c } \hline E & F & G & H \\ \hline \end{array} \end{array} $	A0 = A A1 = B A2 = C A3 = D A4 = E A5 = F A6 = G A7 = H	A0 = D A1 = C A2 = B A3 = A A4 = H A5 = G A6 = F A7 = E
RTS RTSD RTSTK RTSTKD	$ \text{PC} = \begin{array}{ c c c c } \hline & 31 & & 0 \\ \hline A & B & C & D \\ \hline \end{array} $	A0 = A A1 = B A2 = C A3 = D	A0 = D A1 = C A2 = B A3 = A

Table 2-31 shows the representation of the control instructions in big and little endian modes. In this table, it is assumed that the stack access is to address A0.

Table 2-31. Control Instructions in Big and Little Endian Modes

Instruction	Register Operands	Big Endian	Little Endian
TRAP ILLEGAL Interrupt Service	$ \begin{array}{c} \text{PC} = \begin{array}{ c c c c } \hline & 31 & & 0 \\ \hline A & B & C & D \\ \hline \end{array} \\ \text{SR} = \begin{array}{ c c c c } \hline E & F & G & H \\ \hline \end{array} \end{array} $	A0 = A A1 = B A2 = C A3 = D A4 = E A5 = F A6 = G A7 = H	A0 = D A1 = C A2 = B A3 = A A4 = H A5 = G A6 = F A7 = E

Chapter 3

Control Registers

This chapter describes the core control registers for the SC140 core.

Several bits in these registers are not used, and are marked as reserved. These bits are initialized with a zero value and should be written with a zero value for future compatibility.

3.1 Core Control Registers

The SC140 programming model contains two 32-bit core control registers: a status register (SR) and an exception and mode register (EMR). These registers include dedicated bits for reflecting and controlling different operating modes of the core as well as various status flags.

3.1.1 Status Register (SR)

The SR contains 32 bits. It reflects and controls the following:

- Core working mode (Normal or Exception)
- State of the four hardware loops and type of the currently executing loop
- Current interrupt priority level (IPL) of the core
- Overflow exceptions enabled or disabled
- Interrupts enabled or disabled
- Viterbi flags
- Scaling, rounding, and arithmetic saturation modes
- Numeric range of moved data after scaling
- Result (true or false) of a condition test
- Existence of a carry/borrow generated from the last addition/subtraction operation
- Value of last shifted bit during a DALU shift operation

When a subroutine or exception is serviced, the status register is pushed onto the stack. The following instructions implicitly push the SR onto the stack:

- JSR/D
- BSR/D

Any exception or interrupt implicitly pushes the SR onto the stack, including exceptions that are triggered by the following instructions:

- TRAP

- ILLEGAL
- DEBUG, DEBUGEV (if configured in the EOnCE to generate an exception)

The following instructions implicitly pop the SR from the stack:

- RTE/D

Refer to [Appendix A, “SC140 DSP Core Instruction Set,”](#) for a full description of these instructions.

The pipeline imposes certain programming rules relating to the minimum distance between writing the SR and when the change takes effect. For further details, refer to [Chapter 7, “Programming Rules.”](#)

Figure 3-1 shows the bits that make up the status register

	BIT 31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
	SLF	LF3	LF2	LF1	LF0				I2	I1	I0	OVE	DI	EXP		
TYPE	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw
RESET	0	0	0	0	0	0	0	0	1	1	1	0	0	1	0	0
	BIT 15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	BIT 0
					VF3	VF2	VF1	VF0		S	S1	S0	RM	SM	T	C
TYPE	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw
RESET	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Figure 3-1. Status Register -SR

Table 3-1 describes the various SR bits.

Table 3-1. Status Register Description

Name	Description	Settings
SLF Bit 31	Short Loop Flag — Indicates (when set) that the active loop is a short loop, which means that it contains only one or two execution sets. At the start of an interrupt service routine (ISR), the SR (including the SLF bit) is pushed onto the software stack and the SLF bit is cleared. This bit is cleared at core reset.	0 = Active loop length is three or more execution sets 1 = Active loop length is one or two execution sets
LF3 Bit 30	Loop Flag 3 — Indicates (when set) that hardware loop #4 is enabled. At the start of an ISR, the SR (including the LF3 bit) is pushed onto the software stack and the LF3 bit is cleared. This bit is cleared at core reset.	0 = Hardware loop #4 not enabled 1 = Hardware loop #4 enabled

Table 3-1. Status Register Description (Continued)

Name	Description	Settings																																													
LF2 Bit 29	Loop Flag 2 — When set, indicates that hardware loop #3 is enabled. At the start of an ISR, the SR (including the LF2 bit) is pushed onto the software stack and the LF2 bit is cleared. This bit is cleared at core reset.	0 = Hardware loop #3 not enabled 1 = Hardware loop #3 enabled																																													
LF1 Bit 28	Loop Flag 1 — When set, indicates that hardware loop #2 is enabled. At the start of an ISR, the SR (including the LF1 bit) is pushed onto the software stack and the LF1 bit is cleared. This bit is cleared at core reset.	0 = Hardware loop #2 not enabled 1 = Hardware loop #2 enabled																																													
LF0 Bit 27	Loop Flag 0 — When set, indicates that hardware loop #1 is enabled. At the start of an ISR, the SR (including the LF0 bit) is pushed onto the software stack and the LF0 bit is cleared. This bit is cleared at core reset.	0 = Hardware loop #1 not enabled 1 = Hardware loop #1 enabled																																													
R Bits 26–24	Reserved																																														
I2–I0 Bits 23–21	Interrupt Mask Bits — Reflect the current interrupt priority level (IPL) of the core. Only non-maskable interrupts or interrupts with an IPL higher than the current interrupt mask value can interrupt the core. The current IPL of the core can be changed under software control. At the start of an ISR, the SR (including the interrupt mask bits) is pushed onto the software stack. The interrupt mask bits are changed to the IPL of the interrupt being serviced. The interrupt mask bits are set at core reset. For a detailed description of interrupt service, refer to Section 5.8, “Exception Processing,” on page 5-46.	<table border="1"> <thead> <tr> <th>I2</th> <th>I1</th> <th>I0</th> <th>Exceptions Permitted</th> <th>Exceptions Masked</th> </tr> </thead> <tbody> <tr> <td>0</td> <td>0</td> <td>0</td> <td>IPL 1–7</td> <td>IPL 0</td> </tr> <tr> <td>0</td> <td>0</td> <td>1</td> <td>IPL 2–7</td> <td>IPL 0–1</td> </tr> <tr> <td>0</td> <td>1</td> <td>0</td> <td>IPL 3–7</td> <td>IPL 0–2</td> </tr> <tr> <td>0</td> <td>1</td> <td>1</td> <td>IPL 4–7</td> <td>IPL 0–3</td> </tr> <tr> <td>1</td> <td>0</td> <td>0</td> <td>IPL 5–7</td> <td>IPL 0–4</td> </tr> <tr> <td>1</td> <td>0</td> <td>1</td> <td>IPL 6–7</td> <td>IPL 0–5</td> </tr> <tr> <td>1</td> <td>1</td> <td>0</td> <td>IPL 7</td> <td>IPL 0–6</td> </tr> <tr> <td>1</td> <td>1</td> <td>1</td> <td>NMI</td> <td>IPL 0–7</td> </tr> </tbody> </table> <p>An IPL0 exception is always masked.</p>	I2	I1	I0	Exceptions Permitted	Exceptions Masked	0	0	0	IPL 1–7	IPL 0	0	0	1	IPL 2–7	IPL 0–1	0	1	0	IPL 3–7	IPL 0–2	0	1	1	IPL 4–7	IPL 0–3	1	0	0	IPL 5–7	IPL 0–4	1	0	1	IPL 6–7	IPL 0–5	1	1	0	IPL 7	IPL 0–6	1	1	1	NMI	IPL 0–7
I2	I1	I0	Exceptions Permitted	Exceptions Masked																																											
0	0	0	IPL 1–7	IPL 0																																											
0	0	1	IPL 2–7	IPL 0–1																																											
0	1	0	IPL 3–7	IPL 0–2																																											
0	1	1	IPL 4–7	IPL 0–3																																											
1	0	0	IPL 5–7	IPL 0–4																																											
1	0	1	IPL 6–7	IPL 0–5																																											
1	1	0	IPL 7	IPL 0–6																																											
1	1	1	NMI	IPL 0–7																																											


Table 3-1. Status Register Description (Continued)

Name	Description	Settings
OVE Bit 20	Overflow Exception Enable Bit — Enables or disables the generation of an exception caused by an overflow. The DOVF bit in EMR is always set when an overflow occurs. If the OVE bit is set and the DOVF bit is already set, no exception is generated until the DOVF bit is cleared and set again. See Section 3.1.2, “Exception and Mode Register (EMR),” for more information. This bit is cleared at core reset.	0 = Overflow exception generation is disabled 1 = Overflow exception generation is enabled, unless DOVF bit in EMR is already 1
DI Bit 19	Disable Interrupts Bit — When this bit is set, no maskable interrupts are serviced, regardless of the IPL values, which remain unchanged. This bit can be set with the DI instruction, which ensures that interrupts are masked immediately, and can be cleared with the EI instruction. This bit is cleared at core reset.	0 = Interrupts enabled 1 = Interrupts disabled
EXP Bit 18	Exception Mode Bit — Selects the active stack pointer and working mode of the core. This bit is set at core reset.	0 = Normal working mode, active SP is NSP 1 = Exception working mode, active SP is ESP
R Bits 17–12	Reserved	
VF3–VF0 Bits 11–8	Viterbi Flags — Reflect the status of the two parallel conditional transfers in the MAX2VIT instruction. These flags are generally used in conjunction with the VSL instructions. Two Viterbi flags can be independently set or cleared according to the MAX2VIT result. For more information, see MAX2VIT and VSL in Appendix A, “SC140 DSP Core Instruction Set.” These bits are cleared at core reset.	0 = Appropriate 16-bit portion transferred 1 = Appropriate 16-bit portion not transferred
R Bit 7	Reserved	

Table 3-1. Status Register Description (Continued)

Name	Description	Settings																				
S Bit 6	<p>Scaling Bit — Set when moving a result from a data register (D0–D15) to memory using a MOVES (saturated move) instruction. The scaling bit is set when the absolute value of the data that is moved to memory (after scaling and limiting) is greater than or equal to 0.25 and less than 0.75.</p> <p>The logical equations of this bit, if viewed as functions of the data in the register, are dependent on the scaling mode. If limiting occurs during a data register transfer to memory, the scaling bit is not affected. This bit is a sticky bit and it remains set until explicitly cleared. This bit is cleared at core reset.</p>	<table border="1"> <thead> <tr> <th data-bbox="794 321 887 406">S1</th> <th data-bbox="892 321 984 406">S0</th> <th data-bbox="989 321 1150 406">Scaling Mode</th> <th data-bbox="1155 321 1398 406">S Equation</th> </tr> </thead> <tbody> <tr> <td data-bbox="794 412 887 485">0</td> <td data-bbox="892 412 984 485">0</td> <td data-bbox="989 412 1150 485">No scaling</td> <td data-bbox="1155 412 1398 485">S = (D30 XOR D29) OR S (previous)</td> </tr> <tr> <td data-bbox="794 491 887 563">0</td> <td data-bbox="892 491 984 563">1</td> <td data-bbox="989 491 1150 563">Scale down</td> <td data-bbox="1155 491 1398 563">S = (D31 XOR D30) OR S (previous)</td> </tr> <tr> <td data-bbox="794 570 887 642">1</td> <td data-bbox="892 570 984 642">0</td> <td data-bbox="989 570 1150 642">Scale up</td> <td data-bbox="1155 570 1398 642">S = (D29 XOR D28) OR S (previous)</td> </tr> <tr> <td data-bbox="794 649 887 700">1</td> <td data-bbox="892 649 984 700">1</td> <td data-bbox="989 649 1150 700">Reserved</td> <td data-bbox="1155 649 1398 700">S = Undefined</td> </tr> </tbody> </table>	S1	S0	Scaling Mode	S Equation	0	0	No scaling	S = (D30 XOR D29) OR S (previous)	0	1	Scale down	S = (D31 XOR D30) OR S (previous)	1	0	Scale up	S = (D29 XOR D28) OR S (previous)	1	1	Reserved	S = Undefined
S1	S0	Scaling Mode	S Equation																			
0	0	No scaling	S = (D30 XOR D29) OR S (previous)																			
0	1	Scale down	S = (D31 XOR D30) OR S (previous)																			
1	0	Scale up	S = (D29 XOR D28) OR S (previous)																			
1	1	Reserved	S = Undefined																			
S1–S0 Bits 5–4	<p>Scaling Mode Bits — Specify the scaling to be performed in the DALU shifter/limiter as well as the rounding position in the DALU MAC unit.</p> <p>The shifter/limiter scaling mode affects data read from the D0–D15 registers out to the data memory bus using a MOVES instruction. The scaling mode also affects the calculation of the Ln bit for a class of DALU instructions. See Section 2.2.1.5, “Scaling,” and Section 2.2.1.6, “Limiting,” for more information.</p> <p>The scaling mode also affects the MAC rounding bit position. Correct rounding is maintained when different portions of the registers are read out to the data memory buses. For more information, see Section 2.2.2.6, “Rounding Modes.”</p> <p>During arithmetic saturation mode, the scaling bits are ignored for most DALU instructions. See Section 2.2.2.7, “Arithmetic Saturation Mode.”</p> <p>These bits are cleared at the start of an exception service routine as well as at core reset.</p>	<table border="1"> <thead> <tr> <th data-bbox="794 827 868 912">S1</th> <th data-bbox="873 827 965 912">S0</th> <th data-bbox="970 827 1107 912">Rounding Bit</th> <th data-bbox="1112 827 1417 912">Scaling Mode</th> </tr> </thead> <tbody> <tr> <td data-bbox="794 919 868 970">0</td> <td data-bbox="873 919 965 970">0</td> <td data-bbox="970 919 1107 970">15</td> <td data-bbox="1112 919 1417 970">No scaling</td> </tr> <tr> <td data-bbox="794 976 868 1076">0</td> <td data-bbox="873 976 965 1076">1</td> <td data-bbox="970 976 1107 1076">16</td> <td data-bbox="1112 976 1417 1076">Scale down (1-bit Arithmetic Right Shift)</td> </tr> <tr> <td data-bbox="794 1083 868 1155">1</td> <td data-bbox="873 1083 965 1155">0</td> <td data-bbox="970 1083 1107 1155">14</td> <td data-bbox="1112 1083 1417 1155">Scale up (1-bit Arithmetic Left Shift)</td> </tr> <tr> <td data-bbox="794 1161 868 1212">1</td> <td data-bbox="873 1161 965 1212">1</td> <td data-bbox="970 1161 1107 1212">—</td> <td data-bbox="1112 1161 1417 1212">Reserved</td> </tr> </tbody> </table>	S1	S0	Rounding Bit	Scaling Mode	0	0	15	No scaling	0	1	16	Scale down (1-bit Arithmetic Right Shift)	1	0	14	Scale up (1-bit Arithmetic Left Shift)	1	1	—	Reserved
S1	S0	Rounding Bit	Scaling Mode																			
0	0	15	No scaling																			
0	1	16	Scale down (1-bit Arithmetic Right Shift)																			
1	0	14	Scale up (1-bit Arithmetic Left Shift)																			
1	1	—	Reserved																			
RM Bit 3	<p>Rounding Mode Bit — Selects the type of rounding performed by the DALU during arithmetic operations that involve rounding. See Section 2.2.2.6, “Rounding Modes.”</p> <p>This bit is cleared at core reset.</p>	0 = Convergent rounding selected 1 = Two’s complement rounding selected																				

Table 3-1. Status Register Description (Continued)

Name	Description	Settings
SM Bit 2	<p>Arithmetic Saturation Mode — Selects automatic saturation on 32 bits for data arithmetic and logic unit (DALU) results. This bit provides an arithmetic saturation mode for algorithms that do not recognize or cannot take advantage of the extension register. When the arithmetic saturation mode is set, the scaling mode bits are ignored for most instructions. No scaling is performed. Refer to Section 2.2.2.7, “Arithmetic Saturation Mode,” on page 2-25, for details of arithmetic saturation, including the list of instructions affected by arithmetic saturation with or without scaling.</p> <p>Each individual instruction in Appendix A, “SC140 DSP Core Instruction Set,” lists arithmetic saturation as a condition, if appropriate. This bit is cleared at core reset.</p>	0 = Arithmetic saturation mode not selected 1 = Arithmetic saturation mode selected
T Bit 1	<p>True Bit — Indicates whether the condition being tested by a compare or test instruction is true or false. The T-bit is affected by all instructions that check a condition, such as CMPxx, TSTxx, and BMTSTx. The BMTSET.W instruction also sets this bit if a write to memory fails. Conditional instructions (such as JT, JF, BT, BF, IFT, and others) test the T-bit, and execute accordingly. This bit is cleared during core reset as well as at the start of an exception service routine.</p>	0 = Condition tested by compare or test instruction is false 1 = Condition tested by compare or test instruction is true

Table 3-1. Status Register Description (Continued)

Name	Description	Settings
C Bit 0	<p>Carry Bit — Indicates whether a carry is generated from the resulting most significant bits (MSB) of the last addition operation or a borrow generated in the last subtraction operation. The carry or borrow is generated from bit 39 of the result. The carry bit is also affected by DALU bit manipulation as well as rotate and shift instructions. The carry bit usually holds the value of the last shifted bit.</p> <p>If more than one instruction in an execution set affects the carry bit (according to the instruction definition), then the carry bit is updated by the last instruction (in assembly source order) that actually executes, while the other instructions do not affect the carry bit. If no carry-affecting instructions execute, the carry bit is not affected.</p> <p>This bit is cleared during core reset as well as at the start of an exception service routine.</p>	0 = No carry or borrow generated 1 = Carry generated from last addition, or borrow generated from last subtraction

3.1.2 Exception and Mode Register (EMR)

The purpose of the EMR is to reflect and control exception situations in the core. EMR bits reflect memory configuration as well as the servicing of non-maskable interrupts. EMR bits also reflect exception conditions such as:

- DALU overflow
- EOnCE and software debugging access and control
- Illegal execution set
- Illegal instruction opcode

Figure 3-2 displays the bit configuration of the execution and mode register.

	BIT 31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
									GP6	GP5	GP4	GP3	GP2	GP1	GP0	BEM
TYPE	r	r	r	r	r	r	r	r	r	r	r	r	r	r	r	r
RESET	0	0	0	0	0	0	0	0	IO	IO	IO	IO	IO	IO	IO	IO
	BIT 15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	BIT 0
													NMID	DOVF	ILST	ILIN
TYPE	r	r	r	r	r	r	r	r	r	r	r	r	r	r	r	r
RESET	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Figure 3-2. Exception and Mode Register (EMR)

Table 3-2 describes the EMR fields.

Table 3-2. EMR Description

Name	Description	Settings
R Bits 31–24	Reserved	
GP6–GP0 Bits 23–17	General Purpose Flags — Use of these bits is dependent on the state of external pins. Their function is specific to the SoC.	
BEM Bit 16	Big Endian Memory Bit — Indicates big endian or little endian memory configuration. See Section 2.4.1, “SC140 Endian Support,” for more information. This bit is dependent on the state of an external pin. This pin is sampled at core reset.	0 = Little endian configuration 1 = Big endian configuration
R Bits 15–4	Reserved	
NMID Bit 3	Non-maskable Interrupt (NMI) Disable Bit — Set when an NMI service routine enters execution such as when the NMI vector is fetched. While this bit is set, no pending NMI requests are serviced. The bit is cleared by an RTE instruction, or by writing back 1 to it as explained in Section 3.1.2.1, “Clearing EMR Bits.” The NMI bit cannot be set by the user. It is cleared at reset.	0 = No NMI service executing 1 = NMI service executing
DOVF Bit 2	DALU Overflow Bit — Indicates that an overflow from 40 bits occurred during a DALU operation, or that arithmetic saturation occurred in arithmetic saturation mode (overflow from 32 bits). Whenever there is an overflow, an exception is generated if the OVE bit is set in the SR. Until the bit is cleared, no new exceptions are generated. The DOVF bit is a sticky bit. The bit is set if the appropriate exception occurred. It can only be cleared by writing back 1 to it as explained in Section 3.1.2.1, “Clearing EMR Bits.” The DOVF bit cannot be set by the user, only by the hardware. It is cleared at reset. If the OVE bit is set, the clearing operation should only be performed during the overflow exception service routine. Due to pipeline effects, the overflow exception is not serviced immediately after the instruction that caused the overflow condition.	0 = No overflow or arithmetic saturation occurred 1 = Overflow or arithmetic saturation occurred

Table 3-2. EMR Description (Continued)

Name	Description	Settings
ILST Bit 1	<p>Illegal Execution Set — Indicates whether an execution set grouping rule has been violated (for example, more than one opcode dispatched to an execution unit). The ILST bit is a sticky bit. The bit is set if the appropriate exception occurred, and it can only be cleared by the programmer. The clearing operation should only be performed during the illegal exception service routine. This bit is cleared by writing back 1 to it as explained in Section 3.1.2.1, “Clearing EMR Bits.”</p> <p>Whenever an illegal set is detected, an illegal exception is generated. The conditions that set this bit when violated are listed in Section 5.8.5.1.2, “Illegal Execution Set.”</p> <p>ILST is cleared at reset.</p>	0 = No execution set rule violated 1 = Execution set rule violated
ILIN Bit 0	<p>Illegal Instruction — Indicates that one or more of the instruction opcodes received from program memory are not in the SC140 instruction set. “Holes” in operand tables are detected as illegal. Both opcodes for instructions considered reserved and holes in operand tables are determined to be illegal. Whenever an illegal instruction is detected, an illegal exception is generated.</p> <p>The ILIN bit is a sticky bit. It is set if the appropriate exception occurred, and can only be cleared by the user. The bit is cleared by writing back 1 to it, as explained in Section 3.1.2.1, “Clearing EMR Bits.” This clearing operation should only be performed during the illegal exception service routine.</p> <p>ILIN is cleared at reset.</p>	0 = No instruction set violation 1 = One or more opcodes received are not part of the SC140 instruction set



3.1.2.1 Clearing EMR Bits

The ILIN, ILST, DOVF, and NMID bits can only be set by the hardware. These events should be regarded as asynchronous to the program flow given the complex relationship between the events that set these bits and the program flow. These bits are typically cleared by the SW during an exception service routine. DOVF can be cleared outside of an exception service routine for polling usage.

As a programming guideline, the EMR bits should be cleared with great care, to ensure that no information about new events is lost. An EMR bit is cleared by writing back 1 to it using the BMCLR instruction typically inside an exception service routine. Example 3-1 illustrates the use of the BMCLR instruction in the interrupt service routine of an overflow exception, which is activated when DOVF is set.

Example 3-1. Clearing an EMR Bit

```
BMCLR #0xffffb,EMR.L
```

This instruction writes back a zero to every bit in EMR.L except for DOVF, which is written with the same value it contained when it was read. Because DOVF was set to begin with, it is now cleared. Other bits set in EMR.L are not affected. Due to this special behavior, the EMR should not be stored to the stack during a context switch. This ensures that no bits are cleared unintentionally when the EMR is restored.

3.2 PLL and Clock Registers

The SC140 core provides a programming interface to an on-chip phase-locked loop (PLL). The core has two registers that control settings for the PLL as well as clocks, named PCTL0 and PCTL1. The definition and usage of these registers is chip specific. In systems where the PLL is controlled without these registers, they cannot be used as general purpose registers.

Chapter 4

Emulation and Debug (EOnCE)

The SC140 core provides board and chip-level testing capability through two on-chip modules:

- Enhanced on-chip emulation (EOnCE) module
- Joint test action group (JTAG) test access module

These modules are accessed through the JTAG or EOnCE port.

The EOnCE module provides a means of non-intrusive interfacing with the SC140 core and its peripherals, enabling users to examine registers, memory, or on-chip peripherals. Special circuits and dedicated signals on the core are defined, which avoid sacrificing user-accessible on-chip resources. With respect to developing applications for the SC140, the EOnCE provides application developers the following:

- Breakpoints on address ranges
- Breakpoints on data bus values
- Detection of events, which can initiate a number of different actions determined by a developer
- Non-destructive access to the core and its peripherals
- Various means of profiling
- Program tracing

4.1 Debugging System

With the JTAG or EOnCE interface, the user can insert the SC140 core into a target system while retaining debug control. The EOnCE module is used in DSP devices that are based on the SC140 to debug application software in real time. EOnCE is a separate on-chip block that allows non-intrusive interaction with the core. It is accessible through the contents of the JTAG interface signals as well as from the software. The EOnCE module makes it possible to examine the contents of registers, memory, or on-chip peripherals in a special debug environment. This avoids sacrificing user-accessible on-chip resources to perform debugging.

The EOnCE module provides system-level debugging for real-time systems with the ability to:

- Maintain a running log and trace when tasks and interrupts are executed.
- Debug the operation of real-time operating systems (RTOS).

In addition, the EOnCE:

- Reduces system intrusion when debugging in real time.
- Reduces the use of general-purpose peripherals for debugging I/O activities.
- Standardizes the process of system-level debugging across multiple target platforms.
- Provides a rich set of watchpoint features with real-time operation.
- Provides non-intrusive access capability to peripheral registers (for read and write) while in debug state.
- Supports a trace buffer for program flow tracing.
- Provides a programming model accessible during real time by either software or debugging system.

4.2 Overview of the Combined JTAG and EOnCE Interface

The JTAG and EOnCE blocks are tightly coupled. All EOnCE registers are JTAG compliant. Three different programming models are available when using the JTAG and EOnCE interface:

- EOnCE programming model through a host on the JTAG port
- EOnCE programming model through a host from the core software
- JTAG programming model through a host on the JTAG port

Table 4-1 lists the JTAG or EOnCE interface signals.

Table 4-1. JTAG Interface Signal Descriptions

Signal Name	Signal Description
TDI	Test Data Input — Provides a serial input data stream to the JTAG and EOnCE module. It is sampled on the rising edge of the test clock input (TCK), and has an on-chip pull-up resistor.
TDO	Test Data Output — Provides a serial tri-state capable output data stream from the JTAG and EOnCE modules. It is driven in the Shift-IR and Shift-DR controller states of the JTAG state machine. The signal changes on the falling edge of TCK (see below).
TCK	Test Clock Input — Provides a gated clock to synchronize the test logic and shift serial data to and from the JTAG or EOnCE module.
TMS	Test Mode Select Input — Sequences the JTAG controller's state machine. It is sampled on the rising edge of TCK and has an on-chip pull-up resistor.
TRST	Test Reset — Provides a reset signal to the JTAG TAP controller.

4.2.1 Cascading Multiple SC140 EOnCE Modules in a SoC

A typical SC140SoC uses the JTAG TAP controller for standard defined testing compatibilities and for single/multi-core EOnCE control and EOnCE interconnection control. In a multi-core device the EOnCE modules interconnect in a chain and are configured and controlled by the JTAG port (see Figure 4-1).

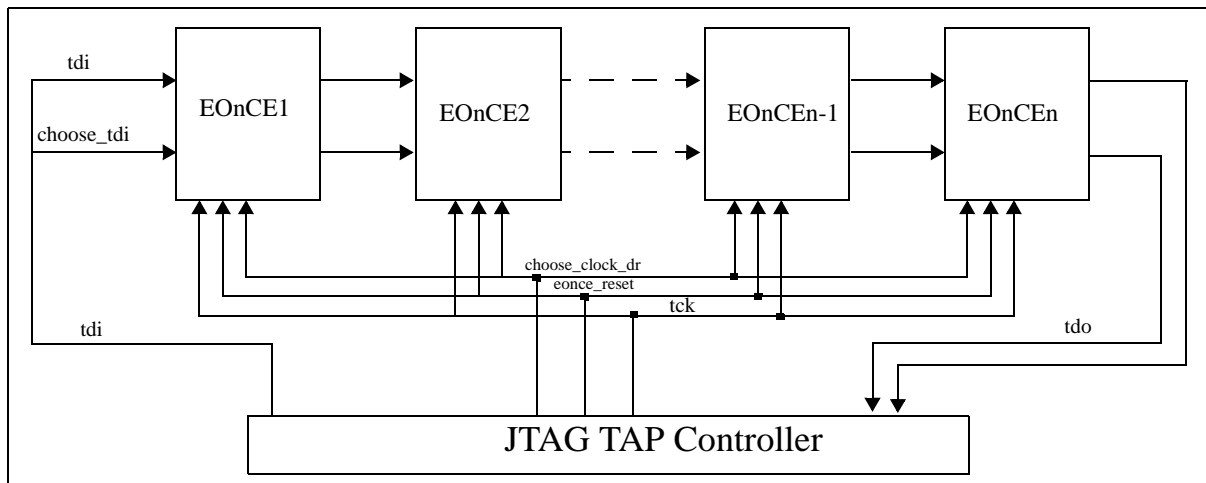


Figure 4-1. JTAG and EOnCE Multi-core Interconnection

To access the EOnCE module of each of the cores through the JTAG port, it is important to know the following:

- The JTAG scan paths
- The JTAG instructions
- The EOnCE control register value

4.2.2 JTAG Scan Paths

The host controller communicates with SoC via the Test Access Port (TAP) controller using the following scan paths:

- *Select-IR JTAG scan path.* Used when the host sends the JTAG instructions shown in **Table 4-2** to the SoC.
- *Select-DR JTAG scan path.* used for data transfer between the HOST and the SoC, which corresponds to the current JTAG instruction exist in the Jtag IR register.

Table 4-2. JTAG Instructions

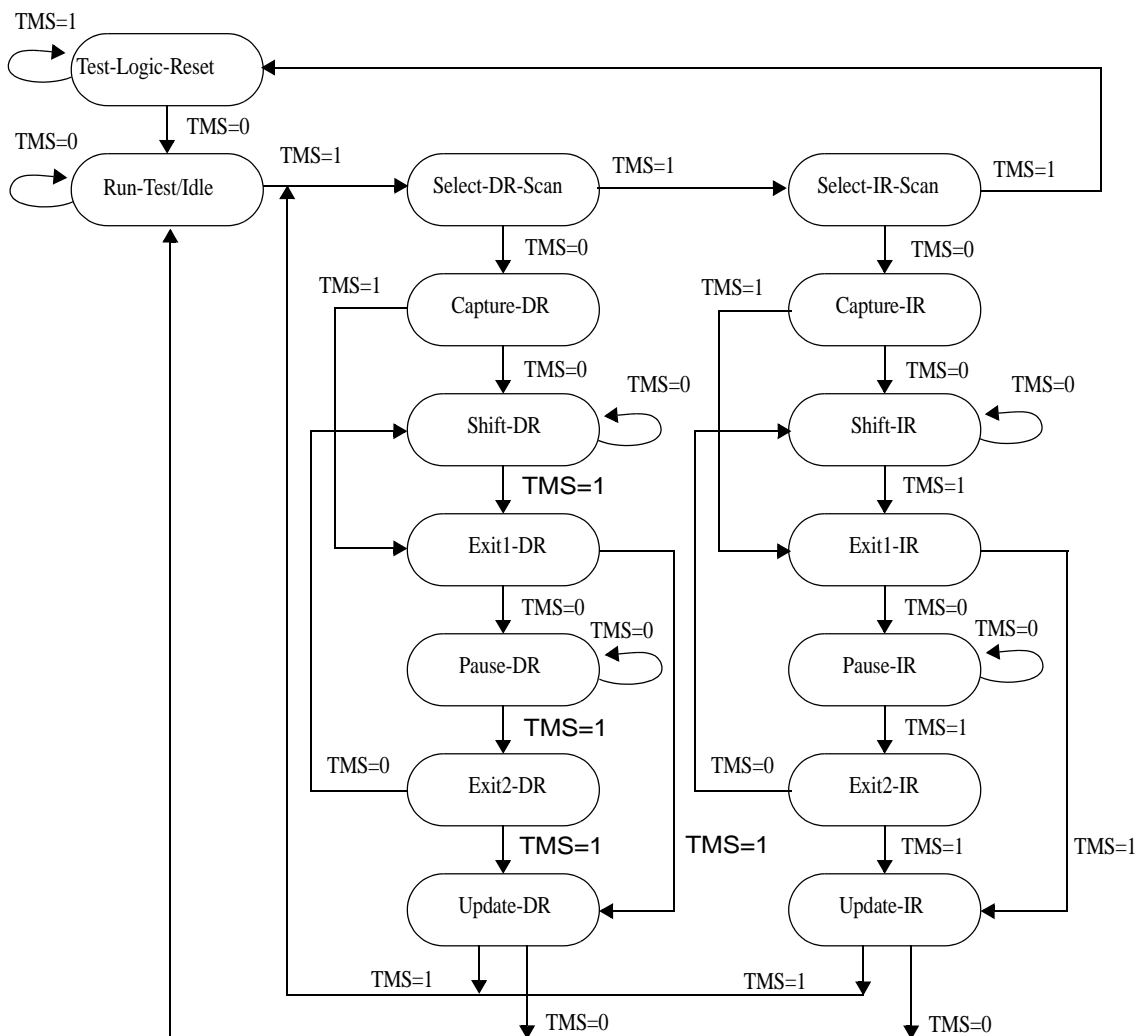
B4	B3	B2	B1	B0	Instruction	Description
0	0	0	0	0	EXTEST	Selects the Boundary Scan Register. Forces a predictable internal state while performing external boundary scan operations.
0	0	0	0	1	SAMPLE/PRELOAD	Selects the Boundary Scan Register. Provides a snapshot of system data and control signals on the rising edge of TCK in the Capture-DR controller state. Initializes the BSR output cells prior to selection of EXTEST or CLAMP.
0	0	0	1	0	IDCODE	Selects the ID Register. Allows the manufacturer, part number and version of a component to be identified.
0	0	0	1	1	CLAMP	Selects the Bypass Register. Allows signals driven from the component pins to be determined from the Boundary Scan Register.



Table 4-2. JTAG Instructions (Continued)

B4	B3	B2	B1	B0	Instruction	Description
0	0	1	0	0	HIGHZ	Selects the Bypass Register. Disables all device output drivers and forces the output to high impedance (tri-state) as per the IEEE specification.
0	0	1	1	0	ENABLE_EONCE	Selects the EOnCE registers. Allows to perform system debug functions. Before this instruction is selected, the CHOOSE_EONCE instruction should be activated to define which EOnCE is going to be activated.
0	0	1	1	1	DEBUG_REQUEST	Selects the EOnCE registers. Forces the chosen cores EOnCE modules into Debug state or generate a Debug exception. Before this instruction, the ENABLE_EONCE and the CHOOSE_EONCE instructions should be performed.
0	1	0	0	0	RUNBIST	Selects the BIST registers. Allows you to generate a built-in self-test for checking the system circuitry.
0	1	0	0	1	CHOOSE_EONCE	Selects the EOnCE registers. Allows to select EOnCE targets in devices with multiple EOnCE modules. This instruction is activated before the ENABLE_EONCE and DEBUG_REQUEST instructions.
0	1	1	0	0	ENABLE_SCAN	Selects the DFT registers. Allows the DFT chain registers to be loaded by a known value or examined in the Shift_DR controller state.
0	1	1	0	1	LOAD_GPR	Allows the component manufacturer to gain access to test features of the device.
0	1	1	1	0	LOAD_SPR	Allows the component manufacturer to gain access to test features of the device.
1	1	1	1	1	BYPASS	Selects the Bypass register. Creates a shift register path from TDI to the Bypass Register and to TDO. Enhances test efficiency when a component other than the current device becomes the device under test.

Figure 4-2 shows the TAP controller state machine, and Table 4-3 shows the states associated with each scan path. The Test Mode Select (TMS) pin determines whether an instruction register scan or a data register scan is performed.


Figure 4-2. TAP Controller State Machine
Table 4-3. JTAG Scan Paths

Select-DR Scan Path	Select-IR Scan Path
Select-DR_SCAN	Select-IR_SCAN
Capture-DR	Capture-IR
Shift-DR	Shift-IR
Exit1-DR	Exit1-IR
Update-DR	Update-IR

At power-up or during normal operation of the host, the TAP is forced into the Test-Logic-Reset state when the TMS signal is driven high for five or more Test Clock (TCK) cycles.

When test access is required, TMS is set low to cause the TAP to exit the Test-Logic-Reset and move through the appropriate states. From the Run-Test/Idle state, an instruction register scan or a data register scan can be issued to transition through the appropriate states.



The first action that occurs when either block is entered is a Capture operation. The Capture-DR state captures the data into the selected serial data path, and the Capture-IR state captures status information into the instruction register. The Exit state follows the Shift state when shifting of instructions or data is complete. The Shift and Exit states follow the Capture state so that test data or status information can be shifted out and new data shifted in. Registers in the selected scan path hold their present state during the Capture and Shift operations. The Update state causes the registers to update with the new data that is shifted into the selected scan path.

4.2.3 Activating the EOnCE Through the JTAG Port

Each of the on chip EOnCE modules has an interface to a JTAG port (via the TAP controller). The interface is active even when a reset signal to the SC140 core is asserted. However, the system reset must be de-asserted to allow proper interface with the cores. This interface is synchronized with internal clocks derived from the JTAG TCK clock. Through the JTAG-EOnCE interface, the JTAG TAP controller can perform the following actions:

- Choose one or more EOnCE blocks (CHOOSE_EONCE instruction)
- Issue a debug request to the EOnCE (DEBUG_REQUEST instruction)
- Enable the chosen EOnCE modules.
- Write an EOnCE command to the EOnCE Command Register.
- Read and write internal EOnCE registers.

4.2.4 Enabling the EOnCE Module

The CHOOSE_EONCE mechanism allows integration of multiple SC140 cores and thus multiple EOnCE modules on the same device. Using the CHOOSE_EONCE instruction, you can selectively activate one or more of the EOnCE modules on the device. The EOnCE modules selected by the CHOOSE_EONCE instruction are cascaded as shown in Figure 4-3. Only selected EOnCE modules respond to ENABLE_EONCE and DEBUG_REQUEST instructions from the JTAG. In Motorola implementations, if the DEBUG_REQUEST instruction is asserted during core reset, until reset de-assertion, all EOnCE modules respond to the instruction and enter debug state when the core leaves reset. However, for driver compatibility with non-Motorola implementations, the JTAG driver should perform CHOOSE_EONCE also during reset for Motorola parts as well. The CHOOSE_EONCE instruction in this case will have no effect. All EOnCE modules are deselected after reset. Since all the EOnCE modules are cascaded, the selection procedure when not in reset is performed serially. The sequence is as follows:

1. Select the CHOOSE_EONCE instruction.
2. At Shift_DR state, enter the serial stream that specifies the modules to be selected.

The number of bits in the serial stream, that is, the number of clocks in this state, is equal to the number of SC140 cores in the cascade. This state is indicated by the CHOOSE_CLOCK_DR signal. To activate the n -th core in the cascade, which is the closest to TDO and the farthest from TDI, the data is 1,0,0,0,...,0 (first a one, then $n-1$ zeros). If the data is 1,0,1,0,0...0 then both the n -th and the $n-2$ th cells are selected.

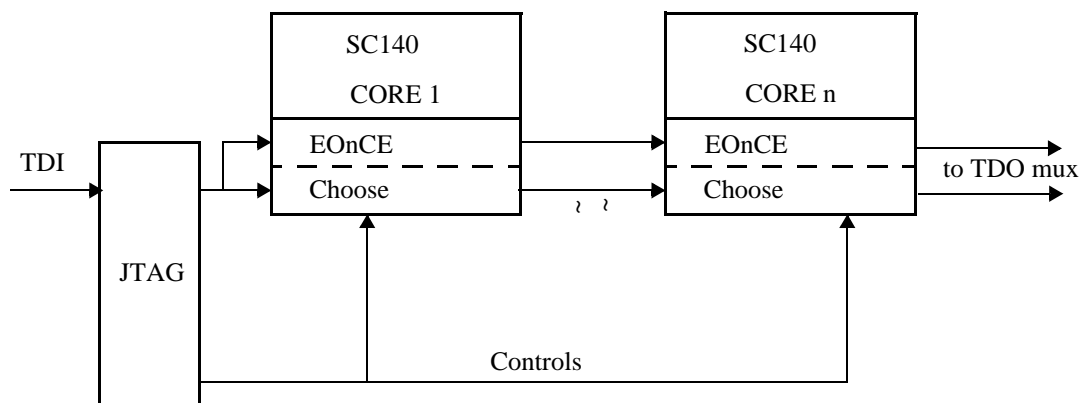


Figure 4-3. Cascading Multiple EOnCE Modules

4.2.5 DEBUG_REQUEST and ENABLE_EONCE Commands

After the CHOOSE_EONCE instruction completes, DEBUG_REQUEST and ENABLE_EONCE instructions can be executed. More than one such instruction can execute, and other instructions can be placed between them and also between them and the CHOOSE_EONCE instruction. The EOnCE modules selected in the CHOOSE_EONCE instruction remain selected until the next CHOOSE_EONCE instruction. The DEBUG_REQUEST or ENABLE_EONCE instruction is shifted in during the Shift-IR state, as are all JTAG instructions.

4.2.6 Reading/Writing EOnCE Registers Through JTAG

An external host can read or write almost every EOnCE register through the JTAG interface, using the following steps (see Figure 4-4):

1. Execute the CHOOSE_EONCE command in the JTAG.
2. Send the data showing which EOnCE is chosen. This command enables the JTAG to manage multiple EOnCE modules in a device.
3. Execute the ENABLE_EONCE command in the JTAG.
4. Write the EOnCE command into the EOnCE Command register (ECR).

That is, the host enters the JTAG TAP state machine into the shift-dr state and then gives the required command on the TDI input signal. After the command is shifted in, the JTAG TAP state machine must enter the update-dr state. The data shifted via the TDI is sampled into the ECR. For example, if the command written into the ECR is “Write EDCA0_CTRL,” then the host must again enter the JTAG into shift-dr and shift the required data, which is to be written into the EDCA0_CTRL, via TDI. If the command is “read some register,” then the DR chain must be passed again and the contents of the register are shifted out through the TDO output.

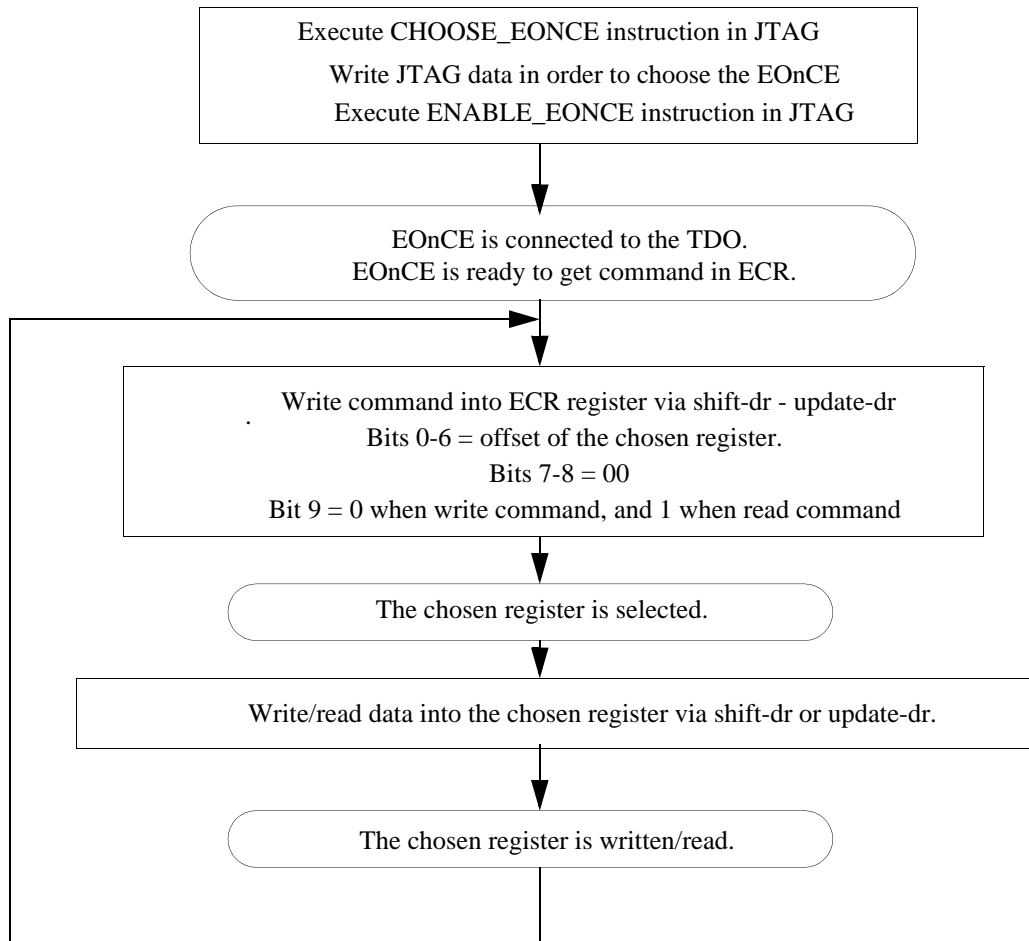


Figure 4-4. Reading and Writing EOnCE Registers Via JTAG

The SC140 EOnCE has several shift registers to interface with the JTAG controller, one shift register per functional unit. Each such shift register is used to interface with all the EOnCE registers in that unit. The length of the each shift register is therefore equal to the length of the longest register in that unit. The list of registers and the shift length is listed in Table 4-12. Figure 4-5 describes how to access a register whose length is shorter than the shift register's length (8 bits versus 32 bits, in this example). In write operations, the last bits to be shifted in the shift register are written to the EOnCE register, the rest have no effect. In read operations, the first bits shifted out of the shift register hold the read data. The rest of the bits are read as zeros.

This organization shortens the access time to the EOnCE registers if the access is only to those registers, and no other devices on the JTAG chain are accessed on that transaction. In such a case, reads and writes can be done only to the actual register length. In case other devices participate in the transaction, the full shift register length should be used, using the convention outlined in Figure 4-5. From the IEEE 1149.1 standard point of view, some of the EOnCE registers have different “read” and “write” views.

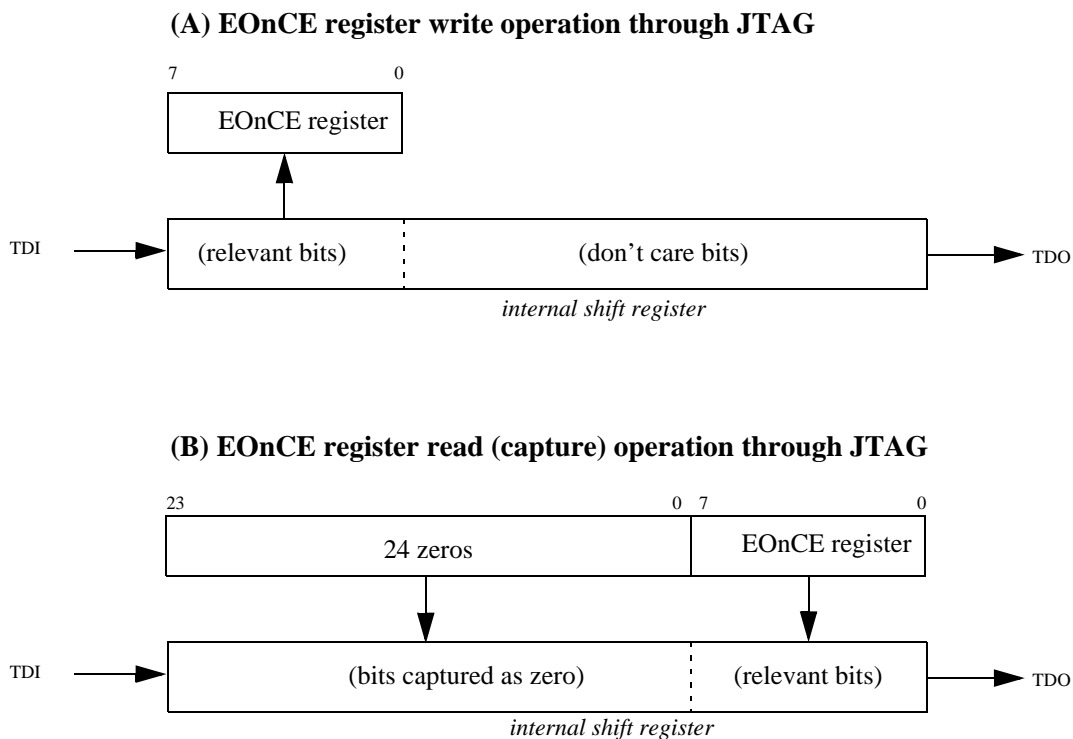


Figure 4-5. Accessing EOnCE registers through JTAG

4.3 Main Capabilities of the EOnCE Module

While the JTAG port provides board test capability, the EOnCE module provides emulation and debug capability. The EOnCE module permits full-speed, real-time, and non-intrusive emulation for a target system or a SC140 development board. This section describes the environment in which the EOnCE module is used for debugging a real-time embedded application. Figure 4-6 shows a typical debug environment where the core resides in a target DSP system.

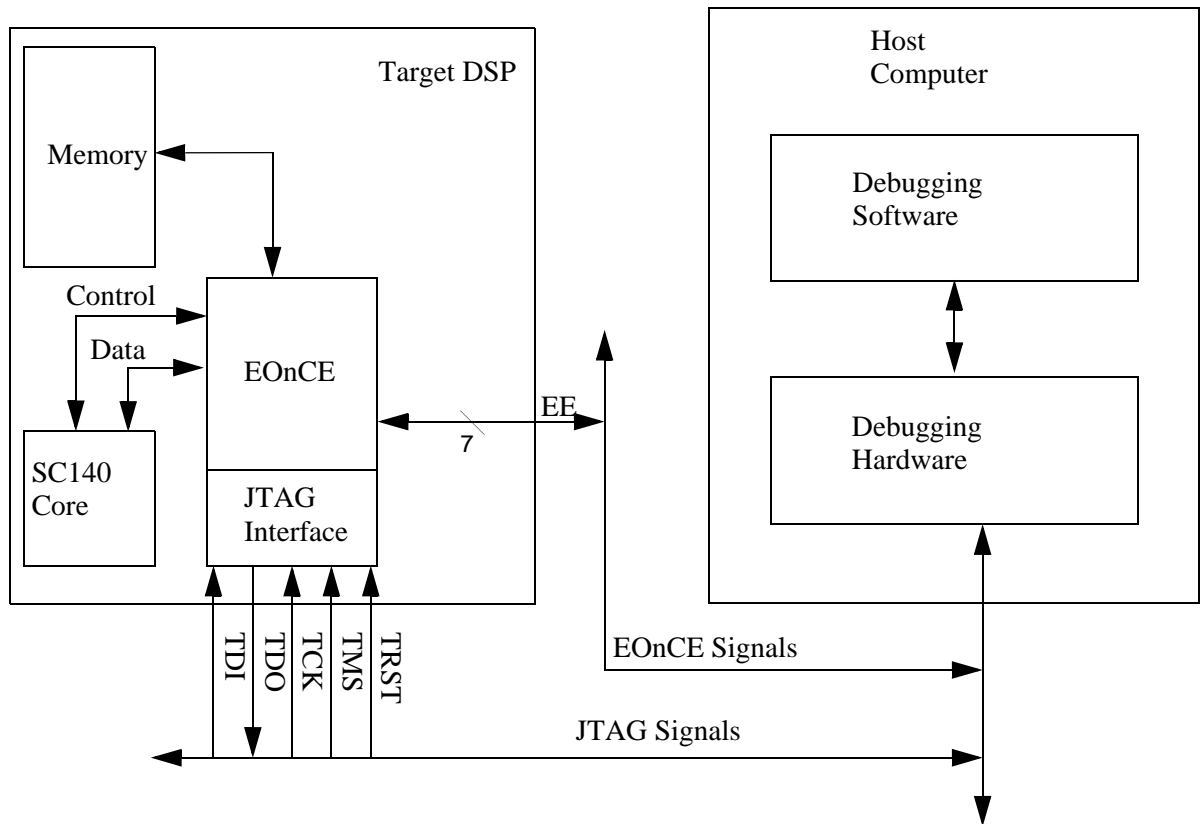


Figure 4-6. Typical Debugging System

4.3.1 EOnCE Signals

The JTAG signals TCK, TDI, and TDO are used to shift data and instructions in and out (see Table 4-1 on page 4-2 for a description of the JTAG signals). For emulation of specific functions, six dedicated EOnCE event signals (EE0–EE5) are available as well as one data event (EED) signal and two event counter (EC) signals.

The two event counter signals EC0 and EC1 allow the event counter to count off-core events such as cache hits/misses, memory contention, external wait-states, etc. These inputs are assumed synchronous to the core clock and support a counting rate up to the core frequency. EC0 and EC1 use is derivative-dependent.

The EE signals can be connected to any on-chip peripheral block such as DMA or TIMER as well as off-chip. This enables an external device to intervene asynchronously in the SC140 debugging process, or to serve as an indication of the events occurring inside the DSP device. Some of these signals have multiple functions programmed by the EE Signals Control Register (EE_CTRL). See [Section 4.7.6, “EE Signals,”](#) for further information.

4.3.2 EOnCE Dedicated Instructions

The instruction set of the SC140 core architecture includes three instructions which are dedicated to the EOnCE module and available for debugging purposes:

- **DEBUG** — Upon decoding by the core, if the SDD bit in EMR is clear, the core enters the debug processing state.
- **DEBUGEV** — This instruction indicates to the EOnCE that a debug event has occurred. The EOnCE handles the instruction according to the settings of the event selector control registers.
- **MARK** — Upon execution by the core when the TMARK bit in the TB_CTRL register is set, its program counter (PC) value is put into the trace buffer. This enables it to mark the different parts of application code that can be executed by different threads. See [Section 4.11.1, “Trace Buffer Control Register \(TB_CTRL\),”](#) for further details.

4.3.3 Debug State

Debug state is a special core processing state in which the pipeline is stalled, waiting for commands from the EOnCE through the JTAG port. All the execution units are ready to operate, but the PSEQ dispatcher module does not dispatch any new execution sets to the execution units. Peripherals can include control bits that determine whether they continue to operate in debug state.

Two actions are possible in debug state:

- **Execute a Single Step** — The core leaves debug state for one cycle. The currently fetched execution set is executed, after which the core then returns to debug state and the PSEQ proceeds to the next execution set.
- **Insert an Instruction from the JTAG port or EOnCE** — A MOVE, JMP, or BRA instruction can be inserted and executed without the core leaving debug state.

The core can be put into debug state by a request from the EOnCE when:

- The DEBUG instruction is issued.
- The EE0 signal is asserted at the exit from reset.
- The EE0 signal is asserted when configured as a debug request (default behavior).
- The JTAG DEBUG_REQUEST instruction is issued at any time, including when the core is exiting reset.
- Assertion a debug request input, to be used for system requests. The usage of this input is SoC specific.
- The trace buffer is full and the TBFDM bit is set in the EOnCE monitor and control register (EMCR).
- The event selector (ES) is programmed to enter the core into debug state upon the detection of an appropriate event.

When the EE0 signal causes the core to enter debug state, the signal must be asserted until the user receives debug acknowledgement.

Asserting the EE0 pin or the JTAG DEBUG_REQUEST instruction signal during reset until getting debug acknowledge will place the core into debug processing state before the first VLES fetch.

If the core is in execution state or in a power-saving state (stop or wait) when a debug request is issued, the core enters debug state. In special cases where the core is frozen (for example, during external access) the core enters debug state after restart of the core clock.

To exit debug state, set the EX bit in the EOnCE command register (ECR) by the EOnCE command shifted through the JTAG port. See [Section 4.7.1, “EOnCE Command Register \(ECR\),”](#) for more details. Debug state is also exited upon a reset.

4.3.4 Debug Exception

Debug exception is a non-maskable core exception, except for the action of the PICINT bit. The PICINT bit in the EMCR register acts as a mode/state switch. If PICINT = 1, a debug event that would otherwise have caused a debug exception asserts instead an EOnCE output to an off-core interrupt controller. If PICINT = 0, the debug event generates a debug exception. This bit is for the use of the system engineer. Exception vectors and priorities are described in [Section 5.8, “Exception Processing,”](#) on page 5-46.

Debug exceptions are generated upon the following:

- The event selector (ES) is programmed to generate a debug exception when an appropriate event occurs.
- The ERCV register is written, and the RCVINT bit in the EMCR register is set.
- The ETRSMT register is read by JTAG, and the TRSINT bit in the EMCR register is set.
- The IME bit in the EMCR register is set, enabling any of the cases that cause the core to enter debug state.

4.3.5 Executing an Instruction while in Debug State

When the core is in debug state, the host connected to the JTAG port can execute a subgroup of the SC140 instruction set in the core. This is done by eliminating the fetch and dispatch stages from the pipeline, and performing only decoding and execution of the instruction directly by an AGU execution unit. The host system writes an instruction to be executed into the core command register (CORE_CMD) together with the GO command. For more information, see [Section 4.7.1, “EOnCE Command Register \(ECR\).”](#)

The subgroup of the instructions that can be executed includes:

- All move instructions with all possible addressing modes
- All types of jump and branch instructions with all possible addressing modes (with the exception of delayed jumps and branches)
- AGU arithmetic instructions

Changes in the state of the core resulting from executing instructions using EOnCE in debug state are the same as when executing the same instructions using core software.

4.3.6 Software Downloading

The JTAG interface along with the EOnCE can be used to download a program into any core-addressable memory. To do this, the CHOOSE_EONCE and DEBUG_REQUEST instructions must have already been executed through the JTAG port, thereby enabling the EOnCE, entering the core into debug state. Figure 4-7 shows a possible flow for software downloading.

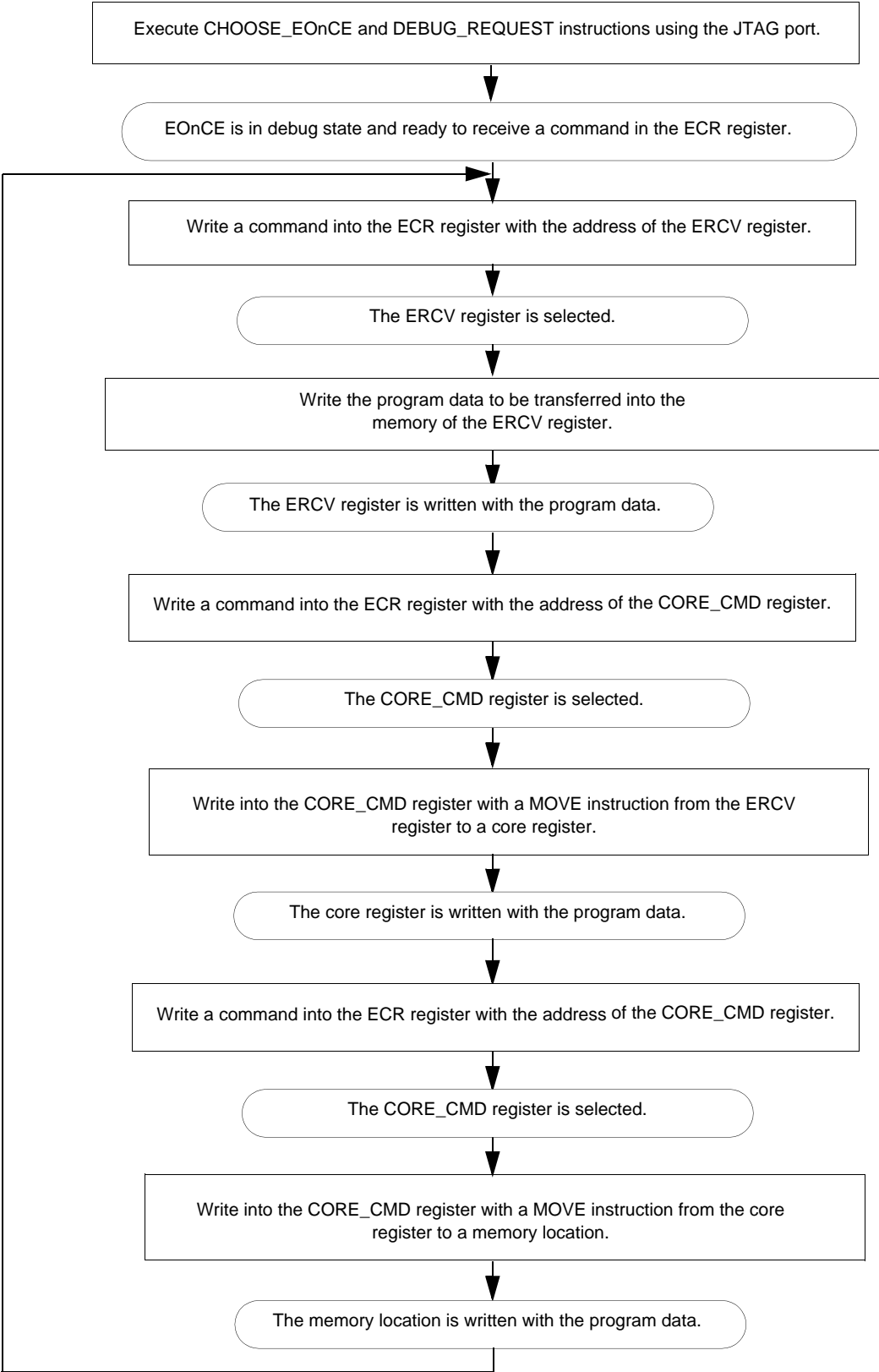


Figure 4-7. Software Downloading

4.3.7 EOnCE Events

An emulator *event* is an occurrence that the emulator can count or trace, or that can cause the emulator to perform an action.

Examples: A core clock cycle is an example of an event because the emulator can count core clock cycles. The execution of a DEBUGEV instruction is another example of an event because the emulator can perform an action—such as placing the core in debug state—whenever the core executes a DEBUGEV instruction.

Table 4-4 below lists EOnCE event types.

Table 4-4. EOnCE Event Types

Event type	Occurs when
DEBUG	The core executes a DEBUG instruction
DEBUGEV	The core executes a DEBUGEV instruction
JTAG DEBUG_REQ	execution of the DEBUG_REQUEST command from the host through the JTAG port
EE	An EE signal (EE0–EE5 or EED) is asserted (when programmed as an input)
Counter	The 31-bit event counter reaches zero
EDCD	The data event-detection channel (EDCD) detects specified values on the data-memory data buses
EDCA	An address event-detection channel (EDCA0–EDCA5) detects specified values on the data-memory address buses or in the program counter
External EDCA	Data address detection events detected by two optional external EDCA channels (EDCA6, EDCA7)
Trace Buffer Full	The trace buffer is full
ERCV	The host writes the most-significant half of the ERCV register
ETRSMT	The host reads the most-significant half of the ETRSMT register
VLES	The core executes a VLES
Clock	A new core clock cycle begins
Trace Transaction	The emulator writes a record to the trace buffer
EC	An EC signal (EC0 or EC1) is asserted
MARK	The core executes a MARK instruction
COF	A change of flow occurs during program execution

4.3.8 EOnCE Actions

An emulator *action* is something that the EOnCE does as a result of an emulator event.

Example: Action Placing the core in debug state is an example of an action.

The EOnCE can perform the following actions:

- Place the core in debug state.
- Generate a debug exception or external interrupt request
- Enable program tracing.
- Disable program tracing.
- Enable the counter.
- Enable the EDCD.
- Enable EDCAs.

4.3.9 Event and Action Summary

Table 4-5 summarizes the events and their possible actions they may cause, depending on the EOnCE programming.

Table 4-5. EOnCE Event and Action Summary

Event type	Counted	trace trigger	Debug state	Debug exception	Enable tracing	Disable tracing	Other actions
DEBUG	-	-	+	+	-	-	
DEBUGEV	+	-	+	+	+	+	
JTAG DEBUG_REQ	-	-	+	+	-	-	
EE	-	-	+	+	+	+	<ul style="list-style-type: none"> • Controls when corresponding EDCD and EDCAs are enabled • Enable the counter
Counter	-	-	+	+	+	+	<ul style="list-style-type: none"> • Enable one or more EDCAs • Enable the EDCD
EDCD	+	-	+	+	+	+	<ul style="list-style-type: none"> • Enable one or more EDCAs • Enable the counter •
EDCA	+	-	+	+	+	+	<ul style="list-style-type: none"> • Enable other EDCAs • Enable the EDCD • Enable the counter
External EDCA	+	-	+	+	+	+	<ul style="list-style-type: none"> • Enable other EDCAs • Enable the EDCD • Enable the counter
Trace Buffer Full	-	-	+	+	-	-	
ERCV	-	-	-	+	-	-	
ETRSMT	-	-	-	+	-	-	
VLES	+	+	-	-	-	-	

Table 4-5. EOnCE Event and Action Summary

Event type	Counted	trace trigger	Debug state	Debug exception	Enable tracing	Disable tracing	Other actions
Clock	+	-	-	-	-	-	
Trace Transaction	+	-	-	-	-	-	
EC	+	-	-	-	-	-	
MARK	-	+	-	-	-	-	
COF	-	+	-	-	-	-	

4.4 EOnCE Enabling and Power Considerations

Except for the EOnCE controller, modules are disabled until one of the following occurs:

- Write access is made to one of the EOnCE registers by the core software.
- Execution of either `ENABLE_EONCE` or `DEBUG_REQUEST` instructions by the host

These events enable all the EOnCE modules, which will result in increased power consumption.

4.5 EOnCE Module Internal Architecture

The EOnCE module is composed of five main sub-units, which performs the following main tasks:

- **EOnCE Controller:** Controls the overall behavior of the EOnCE and allows the JTAG port and core software to read and write the EOnCE registers
- **Event Counter:** Counts various events
- **Event Detection Unit (EDU):** Generates events when it detects predefined values on
 - the data-memory address buses
 - the program counter
 - the data-memory data buses
- **Event Selector:** Controls what action is taken when events or a combination of simultaneous events occurs
- **Trace Unit:** Performs non-intrusive program tracing during program execution

The various EOnCE units include a number of registers. The units, the tasks they perform, and the corresponding registers are described in the sections that follow.

4.5.1 EOnCE Controller

The EOnCE controller performs the following functions:

- Reading and writing EOnCE registers through JTAG

- Reading and writing EOnCE registers from the software
- Real-time JTAG port access
- Real-time data transfer
- Executing instructions while in debug state
- Samples core PC information in various states

Figure 4-8 displays the EOnCE controller block diagram.

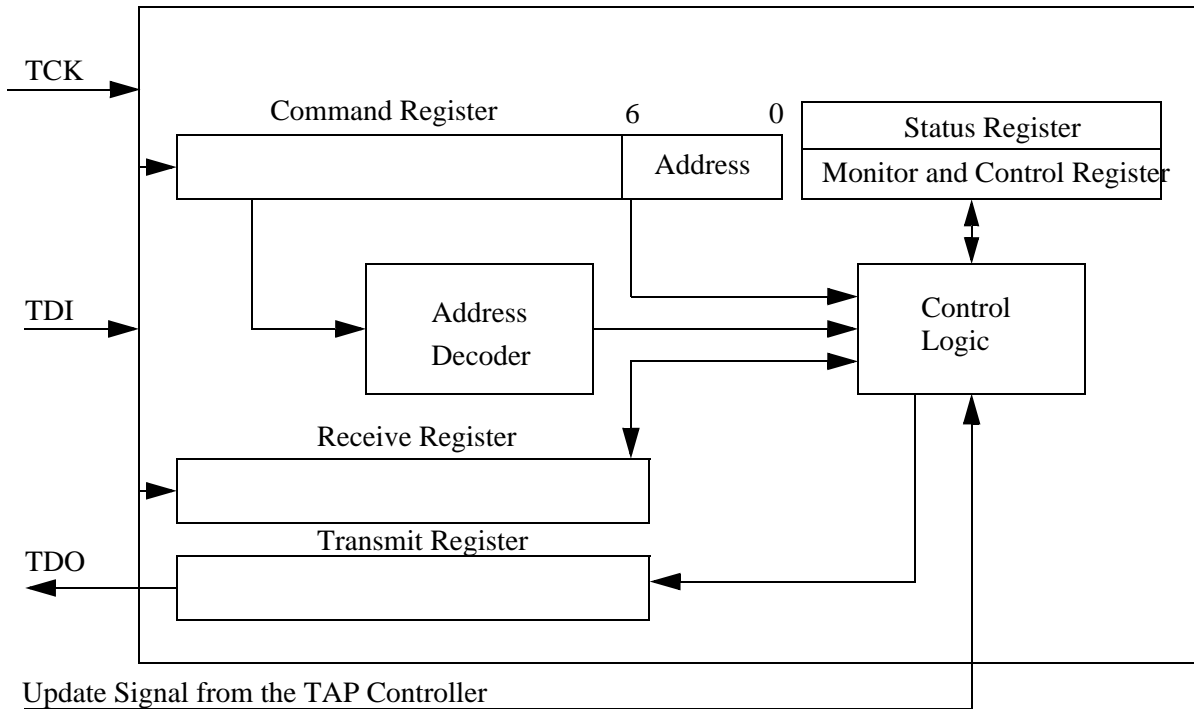


Figure 4-8. EOnCE Controller Block Diagram

The EOnCE controller register set is shown in Table 4-6.

Table 4-6. EOnCE Controller Register Set

Register Name	Description
ECR	EOnCE command register
ESR	EOnCE status register
EMCR	EOnCE monitor and control register
ERCV	EOnCE receive register
ETRSMT	EOnCE transmit register
EE_CTRL	EE signals control register
CORE_CMD	EOnCE core command register
PC_EXCP	PC of the execution set causing illegal or overflow exception

Table 4-6. EOnCE Controller Register Set

Register Name	Description
PC_NEXT	PC of the next execution set
PC_LAST	PC of the last execution set
PC_DETECT	PC breakpoint address register

The functionality of the EOnCE controller registers is described in [Section 4.7, “EOnCE Controller Registers.”](#)

4.5.2 Event Counter

The 64-bit event counter is used to count one of the following possible events:

- System clock
- Instruction execution
- Event detection by an event detection channel
- Tracing into the trace buffer
- Execution of the DEBUGEV instruction
- Off-core events from the EC input signals

When the core is in debug state, the event counter does not count core clocks.

The event counter programming model includes three registers:

- Event counter register (ECNT_CTRL)
- Downcount event counter value register (ECNT_VAL)
- Extension counter value register (ECNT_EXT)

Figure 4-9 shows a block diagram of the event counter.

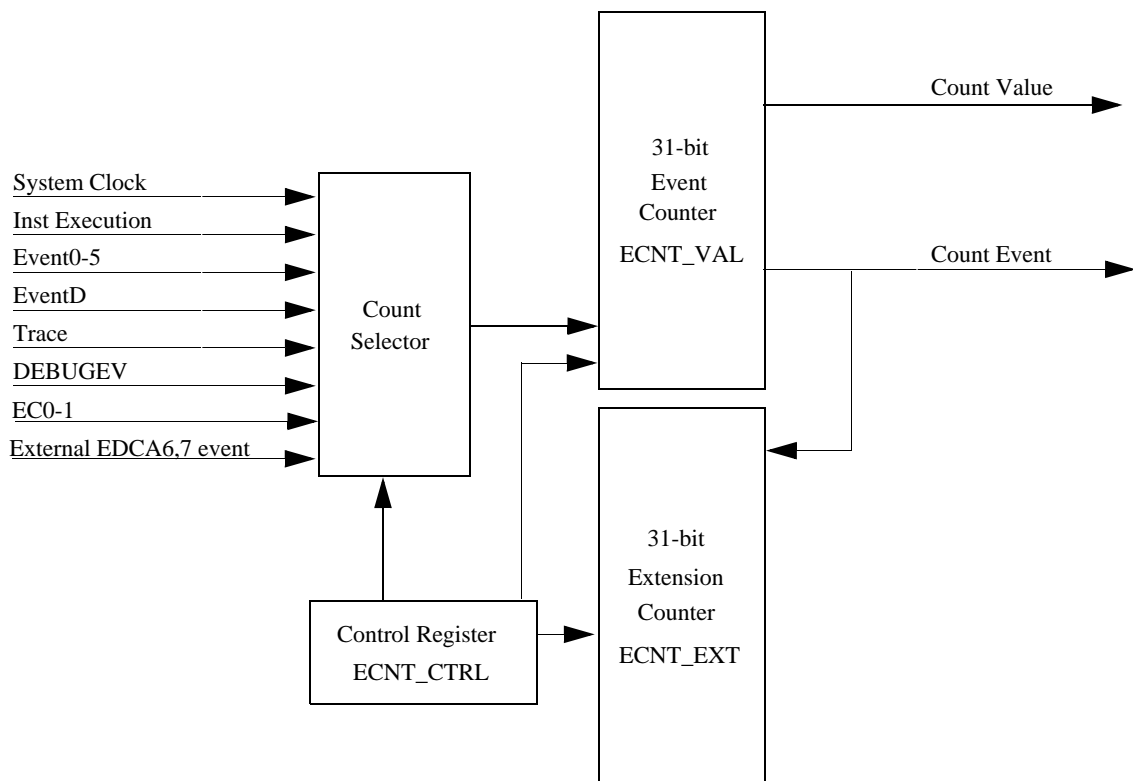


Figure 4-9. Event Counter Block Diagram

ECNT_VAL and ECNT_EXT are 32-bit registers, but their values are limited to 31 bits; their MSB is always zero. Their range is from zero to \$7FFF FFFF. The counter counts down, while the extension counter counts up. The event counter has two counting modes:

- Single count: The counter counts down to zero, and then disables. Upon reaching zero, an EOnCE event is generated (the outcome depends on the event selector).
- Extended count: When the counter reaches zero, it wraps around to \$7FFF FFFF and continues to count. The extension counter is incremented. No EOnCE event is generated.

Table 4-7 shows the event counter register set.

Table 4-7. Event Counter Register Set

Register Name	Description
ECNT_CTRL	Event counter control register
ECNT_VAL	Event counter value register (32-bit)
ECNT_EXT	Extension counter value register (32-bit)

The functionality of the event counter registers is described in [Section 4.8, “Event Counter Registers.”](#)

4.5.3 Event Detection Unit (EDU)

The EOnCE EDU capabilities are:

- Event detection on program and data memory address bus range or value
- Event detection on data memory and data bus range or value
- Detection of data written or read to/from a certain data memory address
- Generating an EOnCE event upon event detection

The EOnCE EDU includes six instances of an Address Event Detection Channel (EDCA), one Data Event Detection Channel (EDCD), and an event selector (ES). In addition, the EDU has an interface that supports adding two additional EDCAs as external modules outside the EOnCE, thus enabling to expand the EOnCE address detection capabilities.

The possible events generated by the EDU are:

- Signal to the event selector (entry to debug state, debug exception, enable or disable tracing) - see [Section 4.5.4, “Event Selector \(ES\).”](#)
- Enable another EDCA or EDCD
- Enable the counter
- Generate a counter event
- Toggle an EE pin (one EE pin assigned to each EDCA or EDCD)

4.5.3.1 Address Event Detection Channel (EDCA)

One of the main elements of the EDU is the EDCA. An EDCA has all the logic required to detect address values according to a user-programmable configuration.

There is no support for breakpoints on the PC of an instruction that is not the first instruction of the execution set. All PC detections are done at execution set level.

Figure 4-11 shows the EDCA block diagram.

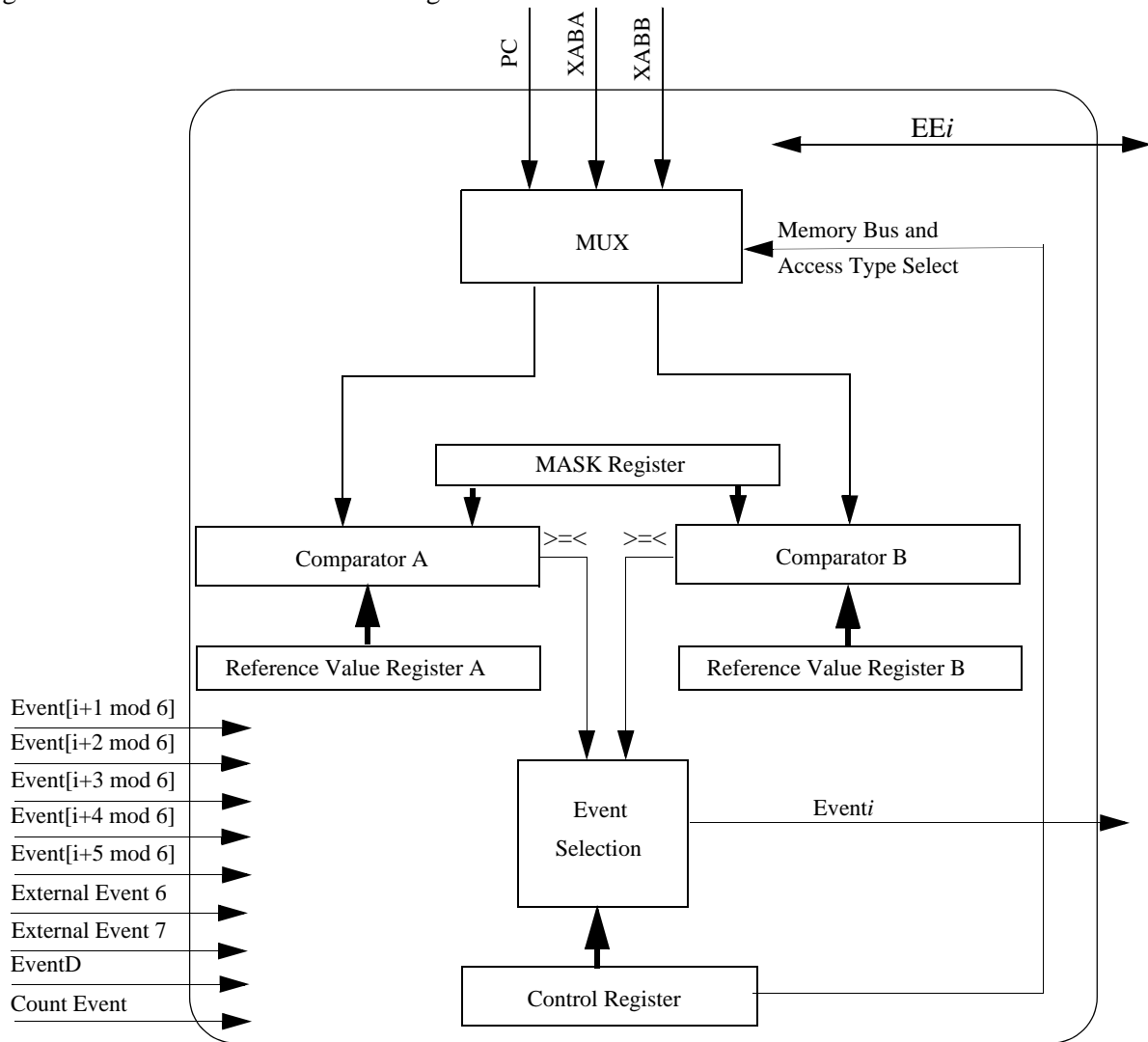


Figure 4-11. EDCA Block Diagram

Two 32-bit comparators are used to compare the core address buses and the reference values programmed into the reference value registers EDCA_i_REFA and EDCA_i_REFB. Each comparator is capable of detecting one of the following four conditions:

- Equal
- Not equal
- Less than

- Greater than

Each EDCA includes four registers, as shown in Table 4-8.

Table 4-8. EDCA Register Set

Register Name	Description
EDCA _i _CTRL	EDCA control register
EDCA _i _REFA	EDCA reference value register A
EDCA _i _REFB	EDCA reference value register B
EDCA _i _MASK	EDCA mask register

The functionality of the EDCA registers is described in [Section 4.9.1, “Address Event Detection Channel \(EDCA\).”](#)

4.5.3.2 Data Event Detection Channel (EDCD)

The EDCCD is one of the main elements of the EDU. It has all the logic required to detect data values according to a user-programmable configuration.

Figure 4-12 shows the EDCCD block diagram.

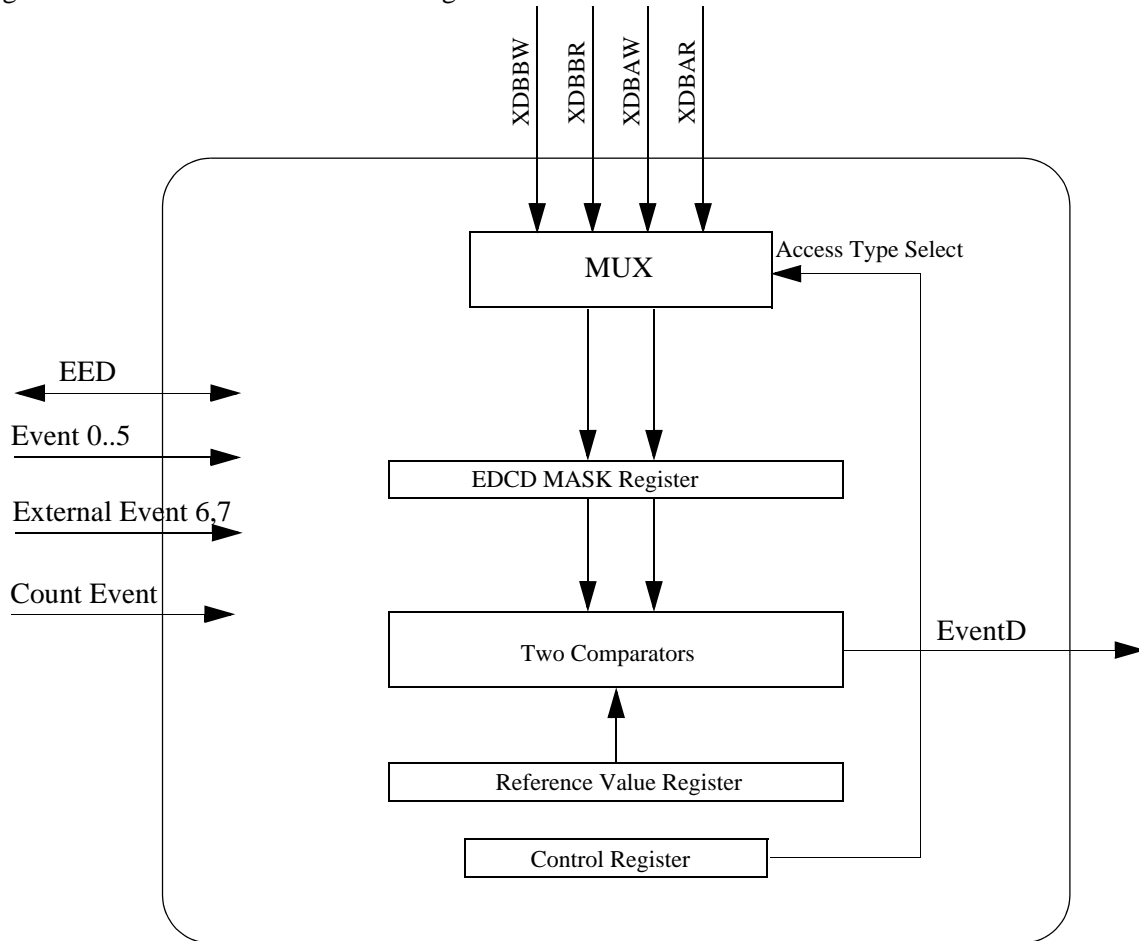


Figure 4-12. EDCCD Block Diagram

The EDCCD register set is shown below.

Table 4-9. EDCCD Register Set

Register Name	Description
EDCCD_CTRL	EDCCD control register
EDCCD_MASK	EDCCD mask register
EDCCD_REF	EDCCD reference value register

The functionality of the EDCCD registers is described in [Section 4.9.2, “Data Event Detection Channel \(EDCCD\).”](#)



4.5.3.3 Optional External Event Detection Address Channels

The EDU has two ports to optional external event detection channels named EDCA6 and EDCA7. If needed, the system designer may add additional event detection capabilities to the EOnCE using these ports. These outcomes of EDCA6/7 events could include the following:

- Placing the core in debug state
- Generating a core debug exception
- Enabling the trace logic
- Disabling the trace logic
- Generating a counter event
- Combinations with other events in the event selector
- Setting the debug reason bits in ESR and the event status bits in EMCR

EDCA6 and EDCA7 do not have an EE pin associated with them.

4.5.4 Event Selector (ES)

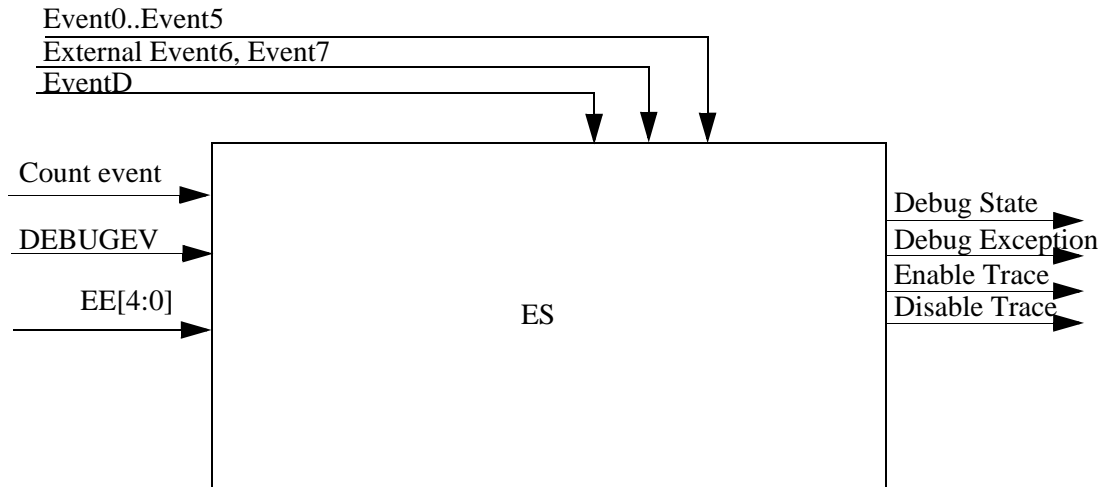
The ES selects the source for various debugging operations. The possible sources that can be selected are:

- Outputs of EDCA instances
- Outputs of optional external EDCA events
- Output of the EDCD
- Output of the event counter
- EE signals
- DEBUGEV instruction

Each of these source events could be programmed¹ in the event selector to cause some or all of the following actions:

- Entry into Debug state
- Execution of a Debug exception
- Enable or disable program tracing

1. They can be programmed individually or combined with other simultaneous events



The ES block diagram is shown in Figure 4-13.

Figure 4-13. Event Selector Block Diagram

The ES can be used to detect reading or writing data from/to a certain data address by using the EDCD to detect the data, an EDCA to detect the address (on XABA, XABB, or both), and the ES to generate an EOnCE event if both events occur. In this case, when both EDCA and EDCD events are selected, only address and data values on the same bus (A or B) can cause an EOnCE event.

Table 4-10 shows the register set of the ES.

Table 4-10. Event Selector Register Set

Register Name	Description
ESEL_CTRL	ES control register
ESEL_DM	ES mask debug state register
ESEL_DI	ES mask debug exception register
ESEL_ETB	ES mask enable trace register
ESEL_DTB	ES mask disable trace register

The functionality of the event selector registers is described in [Section 4.10, “Event Selector \(ES\) Registers.”](#)

4.5.5 Trace Unit

The trace unit is used to store information about a running application without halting its execution. The user can select the addresses to be stored in the trace unit from a wide selection that includes:

- Change-of-flow instructions
 - All Change-of-flow instructions
 - Call/return from subroutine instructions



- Return from exception instructions
- Other change of flow events:
 - Interrupts
 - Hardware loops
- Any execution set
- Mark instructions

For each change-of-flow event, a package of information is stored in the trace buffer, including the PC of the source, the PC of the destination, and, optionally, the value of the event counter and the counter extension.

The EOnCE trace unit:

- Supports a circular hardware trace buffer external to the core. Each entry is 32-bit long. The number of entries is derivative-specific.
- Traces change-of-flow instructions, normal execution, hardware loops, and interrupts.
- Operates during real-time processing.
- Can be read by the debugging hardware during execution state as well as debug state when the trace buffer is disabled.

The trace buffer can be enabled by the host, core software, or by an EOnCE event generated by various ES configurations.

The following addresses can be traced:

- The PC of an execution set containing a taken change-of-flow instruction, followed by its target address.
- The PC of the last execution set executed before servicing an interrupt, followed by the address of the interrupt. See more on interrupt tracing in [Section 4.5.5.1, “Change of Flow and Interrupt Tracing,”](#))
- The PC of every execution set issued.
- The last address for short hardware loops and the last address followed by the start address for long hardware loops.
- The PC of each execution set that includes the MARK instruction.

Figure 4-14 displays a block diagram of the trace unit.

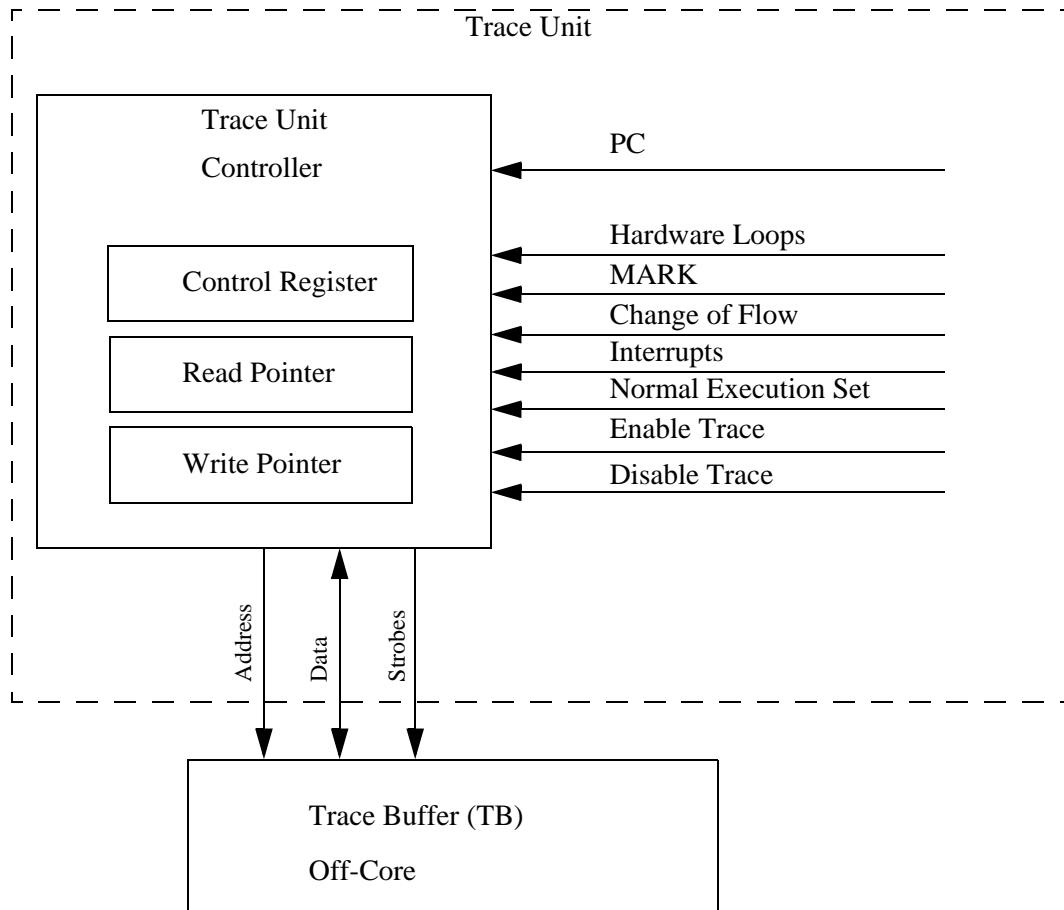


Figure 4-14. Trace Unit Block Diagram

4.5.5.1 Change of Flow and Interrupt Tracing

The trace logic can be configured to trace change of flow instructions. Upon execution of such an instruction, the source and destination addresses of the change of flow event are traced. In case of a delayed change of flow instruction, the source address is also that of the change of flow instruction.

The following change of flow instructions are those that can be traced:

- BT, BF, BTD, BFD
- BRA, BRAD
- JMP JMPD
- JT, JF, JTD, JFD
- JSR, JSRD
- BSR, BSRD
- RTS, RTSD
- RTSTK, RTSTKD
- RTE, RTED

- BREAK
- CONT, CONTD
- SKIPLS

Note that TRAP, and ILLEGAL are traced as interrupts, not as change of flow instructions.

When tracing interrupts, a source destination address pair is also traced. The source address normally reflects the PC of the last executed execution set, and the destination address reflects the PC of the interrupt vector.

There is one exception to this rule: If the interrupt occurred while the core is executing certain instructions (mainly change of flow instructions), it may be that the PC of the execution set including this instruction is traced although not actually executed. This situation is termed “PC KILL”. The debugger SW can identify this case by noting that the return PC upon returning from the exception is that of the killed PC and not that of the following execution set.

4.5.5.2 Writing to the Trace Buffer

The trace buffer is a circular buffer. A write pointer (TB_WR) points to the next free location. The pointer is incremented circularly after every trace, and cleared whenever the trace buffer is enabled.

A flag is set every time the trace buffer is full. The flag is cleared whenever the trace buffer is enabled.

4.5.5.3 Reading the Trace Buffer (TB_BUFF)

The content of the trace buffer is read either through the JTAG interface or from software using the location pointed to by the TB_RD register. The TB_RD pointer is incremented after every read access to the trace buffer, and is cleared when the trace buffer is enabled.

Due to a pre-fetch mechanism, when the user reads the location pointed to by the TB_RD register (by reading the TB_BUFF register), the TB_RD pointer is already three stages ahead. As a result of this pre-fetch mechanism, there is a restriction on reading the trace buffer. A three clock cycle delay must take place from disabling the trace buffer or writing to the read pointer until the first read access is issued to the trace buffer.

The TBFULL bit in the ESR indicates that the buffer is full, and that the contents of the trace buffer should be read. The TBFULL bit of the ESR is set when entry size minus 15 is written. When it reaches the end of the memory, the trace buffer wraps around to address zero and continues until stopped. See the description of the TBFULL bit in [Section 4.7.2, “EOnCE Status Register \(ESR\).”](#)

If the TBFDM in the EMCR is set, and TBFULL is being set, a debug event is generated. If the IME bit in EMCR is clear then the core enters into debug state. If the IME bit is set, a debug exception is generated. This exception can be used by a software routine to empty the trace buffer to an external memory or device. See [Section 4.7.3, “EOnCE Monitor and Control Register \(EMCR\),”](#) and [Section 4.6.1, “Reading or Writing EOnCE Registers Using Core Software,”](#) for further details.

4.5.5.4 Trace Unit Programming Model

The trace unit contains the following registers, as shown in Table 4-11.

Table 4-11. Trace Buffer Register Set

Register Name	Description
TB_CTRL	Trace buffer control register
TB_RD	Trace buffer read pointer register
TB_WR	Trace buffer write pointer register
TB_BUFF	Trace buffer virtual register

The functionality of the trace unit registers is described in [Section 4.11, “Trace Unit Registers.”](#)

4.6 EOnCE Register Addressing

The various units described above use a large number of registers. The EOnCE registers can be read or written when the core is running, or when the core is in debug state.

All the EOnCE registers accessible from the core (either for read or write operations) are memory mapped. This means that each register has its own address in the memory space. The memory address of an EOnCE register is defined by adding four times the register address offset from Table 4-12 on page 4-31 to the EOnCE register base address defined for each SoC derivative. For example, the memory address for the LSB part of register ERVC is $\$8 + rba_via$, where rba_via is the derivative dependent register base address.

Most EOnCE memory-mapped registers allow only 32-bit accesses except the status, monitor, and control registers (ESR, EMCR, EE_CTRL, EDCA[0-5]_CTRL, EDCD_CTRL, ECNT_CTRL, ESEL_CTRL, and TB_CTRL). The latter support 16-bit accesses, which enable the use of bit-mask operations. There is only one access per execution set for all EOnCE registers. When a 16-bit access is used on the 32-bit long ESR and EMCR registers, the software address offset to the MSB part of the registers is equal to the software address offset of the LSB part + 2.

Registers longer than 32 bits are accessed as two registers.

As described in [Section 4.2.6, “Reading/Writing EOnCE Registers Through JTAG,”](#) each EOnCE unit has a shift register supporting the EOnCE registers of this unit. In some cases, the “shift width” of the EOnCE register is longer than its actual width.

Table 4-12 displays the EOnCE register addressing offsets.

Table 4-12. EOnCE Register Addressing Offsets

EOnCE Register Offset	Software Access	Width	Shift width	Register Name	Description
00	R	32	64	ESR	EOnCE status register
01	R/W	32	64	EMCR	Monitor and control register
02	R	64	64	ERCV	EOnCE receive register - least significant part
03	R				EOnCE receive register - most significant part
04	W	64	64	ETRSMT	EOnCE transmit register - least significant part
05	W				EOnCE transmit register - most significant part
06	R/W	16	64	EE_CTRL	EE signals control register
07	R	32	32	PC_EXCP	PC of VLES causing Illegal or Overflow exception
08	NO	32	32	PC_NEXT	PC of next execution set
09	NO	32	32	PC_LAST	PC of last execution set
0A	R	32	32	PC_DETECT	PC breakpoint detection register
.....			Reserved addresses
10	R/W	16	32	EDCA0_CTRL	EDCA0 control register
11	R/W	16	32	EDCA1_CTRL	EDCA1 control register
12	R/W	16	32	EDCA2_CTRL	EDCA2 control register
13	R/W	16	32	EDCA3_CTRL	EDCA3 control register
14	R/W	16	32	EDCA4_CTRL	EDCA4 control register
15	R/W	16	32	EDCA5_CTRL	EDCA5 control register
16				Reserved address	
17				Reserved address	
18	R/W	32	32	EDCA0_REFA	EDCA0 reference value A
19	R/W	32	32	EDCA1_REFA	EDCA1 reference value A
1A	R/W	32	32	EDCA2_REFA	EDCA2 reference value A
1B	R/W	32	32	EDCA3_REFA	EDCA3 reference value A

Table 4-12. EOnCE Register Addressing Offsets (Continued)

EOnCE Register Offset	Software Access	Width	Shift width	Register Name	Description
1C	R/W	32	32	EDCA4_REFA	EDCA4 reference value A
1D	R/W	32	32	EDCA5_REFA	EDCA5 reference value A
1E				Reserved Address	
1F				Reserved Address	
20	R/W	32	32	EDCA0_REFB	EDCA0 reference value B
21	R/W	32	32	EDCA1_REFB	EDCA1 reference value B
22	R/W	32	32	EDCA2_REFB	EDCA2 reference value B
23	R/W	32	32	EDCA3_REFB	EDCA3 reference value B
24	R/W	32	32	EDCA4_REFB	EDCA4 reference value B
25	R/W	32	32	EDCA5_REFB	EDCA5 reference value B
.....			Reserved addresses
30	R/W	32	32	EDCA0_MASK	EDCA0 mask register
31	R/W	32	32	EDCA1_MASK	EDCA1 mask register
32	R/W	32	32	EDCA2_MASK	EDCA2 mask register
33	R/W	32	32	EDCA3_MASK	EDCA3 mask register
34	R/W	32	32	EDCA4_MASK	EDCA4 mask register
35	R/W	32	32	EDCA5_MASK	EDCA5 mask register
36				Reserved address	
37				Reserved address	
38	R/W	16	32	EDCD_CTRL	EDCD control register
39	R/W	32	32	EDCD_REF	EDCD reference value
3A	R/W	32	32	EDCD_MASK	EDCD mask register
.....			Reserved addresses
40	R/W	16	32	ECNT_CTRL	Counter control register
41	R/W	32	32	ECNT_VAL	Counter value register
42	R/W	32	32	ECNT_EXT	Extension counter value
.....			Reserved addresses
48	R/W	8	16	ESEL_CTRL	Selector control register

Table 4-12. EOnCE Register Addressing Offsets (Continued)

EOnCE Register Offset	Software Access	Width	Shift width	Register Name	Description
49	R/W	16	16	ESEL_DM	Selector DM mask
4A	R/W	16	16	ESEL_DI	Selector DI mask
4B				Reserved address	
4C	R/W	16	16	ESEL_ETB	Selector enable TB mask
4D	R/W	16	16	ESEL_DTB	Selector disable TB mask
4E				Reserved address	
4F				Reserved address	
50	R/W	16	32	TB_CTRL	Trace buffer control register
51	R/W	16	32	TB_RD	Trace buffer read pointer
52	R/W	16	32	TB_WR	Trace buffer write pointer
53	R	32	32	TB_BUFF	Trace buffer
.....			Reserved addresses	
7E	NO	48	48	CORE_CMD	Core command register
7F	NO			NOREG	No register selected

4.6.1 Reading or Writing EOnCE Registers Using Core Software

The core can read or write most EOnCE registers from the software. Software access can be disabled by the SWDIS bit in the EMCR register. For more information, see [Section 4.7.3, “EOnCE Monitor and Control Register \(EMCR\).”](#)

In cases where the core is being accessed by the software and the JTAG port at the same time, the JTAG access has priority over the software access.

4.6.2 Real-Time JTAG Access

The EOnCE registers could be read or written to by the host through the JTAG port, as described in [Section 4.2.6, “Reading/Writing EOnCE Registers Through JTAG.”](#)

When the core is not in debug state and the host is accessing the EOnCE registers from the JTAG port, there is a possibility that an EOnCE command may be lost due to a long core stall. To ensure correct execution of a command, the user should read a special ACK bit by shifting out the JTAG IR register together with the core status bits. If the bit is set, this indicates that the last EOnCE command was successfully executed. This bit is reset each time a new command is shifted from the JTAG port to the EOnCE.

The ACK bit could be checked on TDO by executing a “neutral” JTAG EOnCE command such as “ENABLE_EONCE”. Only after it was verified with the ACK bit that the previous access was accepted by the core, the next register could be accessed. This check should be performed also for accessing the ECR.

4.6.3 Real-Time Data Transfer

The EOnCE controller enables the core software to transmit data from the core to the host as well as to receive data sent from the host to the core. This is done by means of a simple receive or transmit mechanism while the core is running.

For transmitting data to the host, the core writes to the transmit register ETRSMT by means of a move instruction using the memory-mapped address of the ETRSMT register. The TRSMT status bit in the ESR is asserted by the EOnCE (see [Section 4.7.2, “EOnCE Status Register \(ESR\),”](#) for more details). The host can poll the TRSMT status bit to see when the data in the ETRSMT register is available. Or alternatively, the host can program the EE4 signal to reflect this status bit externally for interrupt-like transfers, and then read the ETRSMT through TDO using the mechanism described in [Section 4.7.1, “EOnCE Command Register \(ECR\).”](#) The TRSMT bit is cleared by the EOnCE automatically after the ETRSMT register is read by the host. A debug exception can be generated to notify the core that the register can be written again.

The ERCV register can be used for receiving data from the host. The host writes to the ERCV register through the TDI input signal. The EOnCE automatically sets the status bit RCV in the ESR. For more information, see [Section 4.7.2, “EOnCE Status Register \(ESR\).”](#) This bit can be polled by the core to see when the data is ready in the ERCV register, or the application can configure EOnCE to generate a debug exception when the data is ready in the ERCV register. See [Section 4.7.4, “EOnCE Receive Register \(ERCV\),”](#) for more information. The RCV bit is automatically cleared by the EOnCE after the ERCV register is read by the core.

4.6.4 General EOnCE Register Issues

During core reset, the following takes place:

- The selector mask registers are written with zeros.
- All others mask registers are written with ones.
- The EE_CTRL register is written with ones.
- All the remaining registers in the EOnCE programming model are written with zeros.

Only one EOnCE register could be accessed per VLES. It is not allowed to group together a read access from the EOnCE in parallel with an instruction that performs a memory write.

Reserved or unused bits in all registers should be written as zero and the read value should be masked. Writing to unimplemented registers has no effect in the current implementation, but should be avoided for future software compatibility. Reading from unimplemented or write-only registers as well as reading the most significant bits (MSBs) of an 8-bit or 16-bit register with a 32-bit MOVE instruction are both illegal and produces undefined results.

Software write access is possible only if the SWDIS bit in the EMCR register is cleared. The only exception is the ETRSMT register that can always be written by software. When the SWDIS bit is set, read-only access is enabled except reading from the trace buffer.

If the software writes and then reads a given EOnCE register, a NOP or other instruction must be inserted before the read instruction in order to read back the value just written.



Accessibility of the registers through JTAG is the same as from software with the following exceptions:

- The ETRSMT register is only readable only using the JTAG port.
- The ERCV registers are only writable using the JTAG port.
- PC_LAST and PC_NEXT can only be read by the JTAG port.
- The CORE_CMD register can only be written by the JTAG port in debug state.

4.7 EOnCE Controller Registers

A list of the EOnCE controller registers is given in Table 4-6 on page 4-17. The sections that follow describe these registers.

4.7.1 EOnCE Command Register (ECR)

The ECR is a write-only 16-bit shift register that receives its serial data from the TDI input signal. This register is accessed only using JTAG.

Figure 4-15 displays the bit configuration of the ECR.

	BIT 15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	BIT 0
							R/W	GO	EX	REGSEL						
TYPE	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w
RESET	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

The shaded bits are reserved and should be initialized with zeros for future software compatibility.

Figure 4-15. EOnCE Command Register (ECR)

Table 4-13 describes the ECR fields.

Table 4-13. ECR Description

Name	Description	Settings
R Bits 15-10	Reserved	
R/W Bit 9	Read/Write Command — Specifies the direction of data transfer.	0 = Write the data associated with the command into the register specified by REGSEL. 1 = Read the data contained in the register specified by REGSEL.

Table 4-13. ECR Description (Continued)

Name	Description	Settings
GO Bit 8	Go Command — If this bit is set, there are two possible modes of execution: <ul style="list-style-type: none"> When used together with writing or reading a register (except for CORE_CMD), this register is first written or read, and then the next instruction in the pipeline is executed. When used together with the NOREG register, only the next instruction in the pipeline is executed. In this single-step mode, the core leaves debug state for one instruction cycle in order to execute the instruction. If the EX bit is also set, the core continues normal operation after executing the instruction. When used together with writing to the CORE_CMD register, the instruction written to the CORE_CMD register is executed, and the core remains in debug state. If the EX bit is set as well, debug state is exited after the instruction is executed. 	0 = Inactive (no action taken) 1 = Execute one instruction
EX Bit 7	Exit Command — If this bit is set, then after executing any associated write or read command, the core leaves debug state and resumes normal operation. When used together with the write or read NOREG command, the exit command is executed without writing or reading any register.	0 = Remain in debug state. 1 = Exit debug state.
REGSEL Bits 6–0	Register Select — Define which register is the source or destination for the read or write operation. See Table 4-12 on page 4-31 for the EOnCE register offsets.	

4.7.2 EOnCE Status Register (ESR)

The ESR is a 32-bit register. The status bits of the register indicate the status of the core as well as the reason for entering debug state or for issuing a debug exception. All bits are read-only.

Debug reason bits are set to show what caused the core to enter debug state or execute a debug exception. These bits are reset when the core leaves debug state or if the DIS bit in EMCR is reset by an interrupt service routine. After entering debug state, the appropriate bit is set when a new event occurs that could cause the core to enter debug state.

Figure 4-16 displays the bit configuration of the ESR.

	BIT 31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
	CORES			PCKILL	RCV	TRSMT	TBFULL	NOCHOF	REVNO				CORETP			DRTBFULL
TYPE	r	r	r	r	r	r	r	r	r	r	r	r	r	r	r	r
RESET	0	0	0	0	0	0	0	0	x	x	x	0	x	x	x	0

	BIT 15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	BIT 0
	DRSW	DREE 4	DREE 3	DREE 2	DREE 1	DREE 0	DRCOUNTER	DREDCD	DRED CA7	DRED CA6	DRED CA5	DRED CA4	DRED CA3	DRED CA2	DRED CA1	DRED CA0
TYPE	r	r	r	r	r	r	r	r	r	r	r	r	r	r	r	r
RESET	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

The shaded bits are reserved and should be initialized with zeros for future software compatibility. The reset values for REVNO and CORETP (shown as x) are derivative-dependent.

Figure 4-16. EOnCE Status Register (ESR)

Table 4-14 describes the ESR fields.

Table 4-14. ESR Description

Name	Description
CORES Bits 31-30	<p>Core Status — Provides core status information. Indicates whether the core has entered debug state and the reason. These bits are also reflected in the JTAG instruction shift register, which allows for the polling of core status information at the JTAG level. This is useful in case the core software executes a STOP instruction so there are no clocks for reading the core status. The settings for these bits are as follows:</p> <ul style="list-style-type: none"> • 00 = Core is executing instructions. • 01 = Core is in wait or stop mode. • 10 = Core is waiting for a bus. • 11 = Core is in debug state. <p>The “waiting for a bus” state indicates that the core is waiting for data on the bus to be transferred, or that the core is disabled by the external system (for example during memory BIST).</p> <p>The core could be waiting for a bus in any of the processing states. In such a case the CORES field will show “waiting for a bus”.</p>
R Bit 29	Reserved
PCKILL Bit 28	<p>PC Killed — This bit signifies that the last executed VLES was aborted by a pending exception. It is set by the EOnCE when the last execution set has been aborted, and cleared when starting to perform the next single step. This bit is valid only in debug state, and is useful particularly while single stepping.</p>
RCV Bit 27	<p>Receive — Set by the EOnCE when the host has finished writing to the ERCV register. The bit is cleared by EOnCE when both halves of the ERCV register contents are read by the core. The two halves are read in a specific order with the LSB read first. The RCV bit is cleared when the MSB has been read without checking if the LSB part has been read.</p>
TRSMT Bit 26	<p>Transmit — Set by the EOnCE when both halves of the ETSMT register are written by the core. The two halves are written in a specific order with the LSB written first. The TRSMT bit is set when the MSB has been written without checking if the LSB part has been written. The bit is cleared by EOnCE when the host has finished reading the content of the ETRSMT register.</p>

Table 4-14. ESR Description (Continued)

Name	Description
TBFULL Bit 25	Trace Buffer Full — Indicates that the trace buffer of EOnCE is full. In order not to lose addresses when TBFDM and IME bits in the EMCR register are set (when TB is full), the bit causes a debug exception. The TBFULL bit is set when the TB write pointer equals TB-size minus 15, where TB-size is defined for each SoC derivative. TB-size is the size of the off-core trace buffer memory and is defined by the value of 16 core external signals. The TBFULL bit is reset when the trace buffer is enabled. For more information, see Section 4.5.5.3, “Reading the Trace Buffer (TB_BUFF).”
NOCHOF Bit 24	No Change-of-Flow (COF) in Debug State — If this read-only status bit is set by EOnCE upon entering debug state, users cannot inject a change-of-flow instruction through the EOnCE to the core. This occurs when the core is in the following locations: <ul style="list-style-type: none"> • Immediately after executing a delayed change of flow instruction (see Section 7.4.9) • At the end of a long loop, right after executing LA-2 or LA-1 (see Section 5.4.2) • During a short loop. Single-step operations can be used in order to exit this state. When debug state is entered in the middle of a short loop, the loop counter (LC) should be reset and some single-step operations should be executed before injecting a JMP instruction. If this bit is set right before intended return to execution (with the EX bit in ECR), one single-step should be performed before exiting debug state.
REVNO Bits 23–21	Revision Number — The REVNO field generally identifies the basic instruction set revision of the core. It identifies the availability of new instructions and corrections to existing instructions along a binary upward compatible roadmap. Changes in REVNO imply a software tools switch, different software simulator and different host debugger. Cores of different revisions can differ in their EOnCE programming model.
R Bit 20	Reserved
CORETP Bits 19–17	Core Type — The CORETP field identifies the architecture member within the SC100 family. It identifies the availability of new execution units and VLES grouping capabilities. Note that execution units and VLES can scale up or down without altering the basic instruction set. Changes in CORETP imply a software tools switch, different software simulator, and different host debugger.
DRTBFULL Bit 16	Debug Reason is Trace Buffer — Set when the core enters debug state or executes a debug exception as a result of the EOnCE trace buffer being full (TBFULL set). It is cleared by EOnCE when the core exits debug state, or when the DIS bit in EMCR is reset by the user.
DRSW Bit 15	Debug Reason is Software Debug — Set when the core enters debug state or executes a debug exception as a result of the execution of a debug instruction in the core. DRSW is also set when an execution of the DEBUGEV instruction puts the DSP into debug state. It is cleared by EOnCE when the core exits debug state, or when the DIS bit in EMCR is reset by the user.
DREE4 Bit 14	Debug Reason is EE4 — Set when the core enters debug state or executes a debug exception as a result of EE4 assertion. It is cleared by the EOnCE when the core exits debug state, or when the DIS bit in EMCR is reset by the user.

Table 4-14. ESR Description (Continued)

Name	Description
DREE3 Bit 13	Debug Reason is EE3 — Set when the core enters debug state or executes a debug exception as a result of EE3 assertion. It is cleared by the EOnCE when the core exits debug state, or when the DIS bit in EMCR is reset by the user.
DREE2 Bit 12	Debug Reason is EE2 — Set when the core enters debug state or executes a debug exception as a result of EE2 assertion. It is cleared by the EOnCE when the core exits debug state, or when the DIS bit in EMCR is reset by the user.
DREE1 Bit 11	Debug Reason is EE1 — Set when the core enters debug state or executes a debug exception as a result of the EE1 assertion. It is cleared by the EOnCE when the core exits debug state, or when the DIS bit in EMCR is reset by the user.
DREE0 Bit 10	Debug Reason is EE0 — Set when the core enters debug state or executes a debug exception as a result of EE0 assertion. It is cleared by the EOnCE when the core exits debug state, or when the DIS bit in EMCR is reset by the user.
DRCOUNTER Bit 9	Debug Reason is Counter — Set when the core enters debug state or executes a debug exception as a result of a count event. It is cleared by the EOnCE when the core exits debug state, or when the DIS bit in EMCR is reset by the user.
DREDCAD Bit 8	Debug Reason is EDCA7 — Set when the core enters debug state or executes a debug exception as a result of detection by the EDCA7. It is cleared by the EOnCE when the core exits debug state, or when the DIS bit in EMCR is reset by the user.
DREDCA7 Bit 7	Debug Reason is EDCA6 — Set when the core enters debug state or executes a debug exception as a result of detection by the optional external EDCA6. It is cleared by the EOnCE when the core exits debug state, or when the DIS bit in EMCR is reset by the user.
DREDCA6 Bit 6	Debug Reason is EDCA5 — Set when the core enters debug state or executes a debug exception as a result of detection by the optional external EDCA5. It is cleared by the EOnCE when the core exits debug state, or when the DIS bit in EMCR is reset by the user.
DREDCA5 Bit 5	Debug Reason is EDCA4 — Set when the core enters debug state or executes a debug exception as a result of detection by EDCA4. It is cleared by the EOnCE when the core exits debug state, or when the DIS bit in EMCR is reset by the user.
DREDCA4 Bit 4	Debug Reason is EDCA3 — Set when the core enters debug state or executes a debug exception as a result of detection by EDCA3. It is cleared by the EOnCE when the core exits debug state, or when the DIS bit in EMCR is reset by the user.
DREDCA3 Bit 3	Debug Reason is EDCA2 — Set when the core enters debug state or executes a debug exception as a result of detection by EDCA2. It is cleared by the EOnCE when the core exits debug state, or when the DIS bit in EMCR is reset by the user.
DREDCA2 Bit 2	Debug Reason is EDCA1 — Set when the core enters debug state or executes a debug exception as a result of detection by EDCA1. It is cleared by the EOnCE when the core exits debug state, or when the DIS bit in EMCR is reset by the user.
DREDCA1 Bit 1	Debug Reason is EDCA0 — Set when the core enters debug state or executes a debug exception as a result of detection by EDCA0. It is cleared by the EOnCE when the core exits debug state, or when the DIS bit in EMCR is reset by the user.
DREDCA0 Bit 0	Debug Reason is EDCA0 — Set when the core enters debug state or executes a debug exception as a result of detection by EDCA0. It is cleared by the EOnCE when the core exits debug state, or when the DIS bit in EMCR is reset by the user.

4.7.3 EOnCE Monitor and Control Register (EMCR)

The EMCR is a 32-bit register. Bits 31–16 are read/write control bits. Bits 15–0 are sticky status bits and can only be written with zeros. Writing them with a one has no effect. The sticky status bits of the register indicate an event generated by the EOnCE EDU.

Figure 4-17 displays the configuration of EMCR.

BIT	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
							PICINT	TRSINT	TBFD	RCVINT	DEBUGERST			SWDIS	IME	
TYPE	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw
RESET	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

BIT	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	BIT 0
	DIS							EDCD ST	EDCA ST7	EDCA ST6	EDCA ST5	EDCA ST4	EDCA ST3	EDCA ST2	EDCA ST1	EDCA ST0
TYPE	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw
RESET	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

The shaded bits are reserved and should be initialized with zeros for future software compatibility.

Figure 4-17. EOnCE Monitor and Control Register (EMCR)

Table 4-15 describes the EMCR fields.

Table 4-15. EMCR Description

Name	Description
R Bits 31–26	Reserved
PICINT Bit 25	PIC interrupt - When set, and there is a condition that would have caused a debug exception, the EOnCE will assert an external pin to an external interrupt controller. When reset, the core generates an internal debug exception for the same event. This bit is for the use of the system engineer.
TRSINT Bit 24	Transmit Interrupt — Can be set for interrupt driven data messaging. If this bit is set and the TRSMT bit is reset by the EOnCE, a debug exception is issued. The core ISR determines the reason for the interrupt and writes the new data to the ETRSMT register.
TBFD Bit 23	Enter Debug on Trace Buffer Full — When TBFD is cleared, the trace buffer wraps around when full and does not affect core execution. After write pointer wrap-around, the trace buffer over-writes the oldest entries like a modulo buffer. When TBFD is set, the trace buffer almost-full condition affects core execution according to the IME bit. For this condition, IME cleared causes the core to enter debug state and IME set causes the core to take the debug exception. TBFD is cleared on RESET
RCVINT Bit 22	Receive Interrupt — Can be set by the user for interrupt driven data messaging from the host to the target. If this bit is set and the RCV bit is set by the EOnCE, a debug exception is issued. The core interrupt service routine (ISR) determines the reason for the interrupt and reads the content of the ERCV register.

Table 4-15. EMCR Description (Continued)

Name	Description
DEBUGERST Bits 21–18	Debugger Status Information — If several applications (debugger processes) try to connect to the core, unaware of each other, DEBUGERST bits serve as flags. Reset once the core is powered, they can be set/reset by the application as an occupy signal. The debugger may use these bits to reserve the core for its use. In case the host disconnects from the core or goes down, when the host (debugger) tries to regain control on the core, it can use the DEBUGERST bits to find out at when the host disconnected. This is extremely useful when the host is connected to the core through a network rather than direct cables.
SWDIS Bit 17	Software Access Disable — Enables the debug host to lock the EOnCE. When the bit is set, software write access is denied to all the EOnCE registers except the ETRSMT register. Software read access is denied from the trace buffer.
IME Bit 16	Interrupt Mode Enable — When set, this bit causes the core to execute a debug exception instead of entering debug state for any of the source events that would have put the core in debug state. This bit can only be changed when all debug request sources are disabled, specifically when there are no debug requests from the external source (JTAG port, EE pin or system debug request), trace buffer, event selector or from the execution of a debug instruction. Debug request signals from external sources should not normally be used as a source for debug exceptions. If they are used, the interrupt request should be kept asserted until the core acknowledges it to the driver by some agreed SW protocol. The core then must acknowledge that the interrupt was de-asserted before the driver may assert it again.
DIS Bit 15	Debug Interrupt Status — Sticky bit that is set by the EOnCE when a debug exception is generated. When a user resets this bit, all the debug reason bits of the ESR are reset.
R Bits 14–9	Reserved
EDCDST Bit 8	EDCD Status — Sticky bit that is set by the EOnCE upon event detection by the EDCD. Should be cleared by the user.
EDCAST7 Bit 7	EDCA7 Status — Sticky bit that is set by the EOnCE upon event detection by the optional external EDCA7. It should be cleared by the user.
EDCAST6 Bit 6	EDCA6 Status — Sticky bit that is set by the EOnCE upon event detection by the optional external EDCA6. It should be cleared by the user.
EDCAST5 Bit 5	EDCA5 Status — Sticky bit that is set by the EOnCE upon event detection by EDCA5. It should be cleared by the user.
EDCAST4 Bit 4	EDCA4 Status — Sticky bit that is set by the EOnCE upon event detection by EDCA4. It should be cleared by the user.
EDCAST3 Bit 3	EDCA3 Status — Sticky bit that is set by the EOnCE upon event detection by EDCA3. It should be cleared by the user.
EDCAST2 Bit 2	EDCA2 Status — Sticky bit that is set by the EOnCE upon event detection by EDCA2. It should be cleared by the user.
EDCAST1 Bit 1	EDCA1 Status — Sticky bit that is set by the EOnCE upon event detection by EDCA1. It should be cleared by the user.
EDCAST0 Bit 0	EDCA0 Status — Sticky bit that is set by the EOnCE upon event detection by EDCA0. It should be cleared by the user.

4.7.4 EOnCE Receive Register (ERCV)

ERCV is a 64-bit shift register that can be written from the TDI input signal. The register can be read by the software as two 32-bit registers. The ERCV register has to be read in a specific order with the Least Significant Part first. The Least Significant Part read is optional, but the Most Significant Part read is required to clear the RCV bit in the ESR.

ERCV is used to transfer data from the host. This can be done in the following sequence:

1. The host issues a write command to the ERCV register.
2. The host transmits 64 bits through TDI into the ERCV register.
3. The RCV bit in the ESR is set by the EOnCE.
4. If the RCVINT bit in the EMCR is set, the core is interrupted by a debug exception. Otherwise, the core must poll the RCV status bit to know when the data is ready in the ERCV register.
5. The core reads the ERCV register using move instructions.
6. The RCV bit in ESR is cleared by EOnCE. The EE3 signal can be programmed to reflect the value of the RCV bit, informing the host when further data can be transmitted.

4.7.5 EOnCE Transmit Register (ETRSMT)

ETRSMT is a 64-bit shift register that can be read by the TDO output signal. The register can be written by software as two 32-bit registers. The ETRSMT register must be written in a specific order, with the Least Significant Part first. The Least Significant Part write is optional, but the Most Significant Part write is required to set the TRSMT status bit.

The ETRSMT register can transmit data from the core to an external host while the core is running. This can be done in the following sequence:

1. The core writes data to be transmitted into the ETRSMT register.
2. The TRSMT bit in the ESR is set automatically by the EOnCE.
3. The host polls the TRSMT bit in the ESR to detect that the data in the ETRSMT register is available. Alternatively, the host can program the EE4 signal to be set when the TRSMT bit is set.
4. The host issues a read command to the ETRSMT register and reads the register serially through the TDO line.
5. The TRSMT bit is cleared on completion of the read by the host debugger. If the TRSINT bit in the EMCR is set, the core is interrupted by a debug exception, informing the core that further data can be transmitted.

4.7.6 EE Signals

EE signals are general-purpose core interfaces which serve as input or output to the EOnCE. They can be connected off-chip or to a specific on-chip peripheral. This connection is defined by the SoC derivative. In some systems, the EE signals are not connected to an external signal.

4.7.6.1 EE Signals as Outputs

EE signals can be used to indicate internal EOnCE events to devices outside the core. The internal signals which can be indicated are:

- Detection by the event detection channels
- Detection of entry into debug state
- Status bit of the ERVC register
- Status bit of the ETRSMT register

4.7.6.1.1 Detection by the Event Detection Channels

Each EE signal can be configured to serve as an off-core indication of an event detected by the corresponding EDCA or by EDCD. The EE signals in this case work as a toggle. This capability can be used in the following manner:

- One or more event detection channels of the EOnCE can be programmed to detect certain events.
- Each event detection channel toggles its EE signal when the detection of the desired event occurs.
- The time elapsed between the two detected events can be measured by connecting the EE signals to a logic analyzer.

If the EE pin is connected to an I/O pad of the chip, there may be limitations on the frequency of events that could be reflected on these pins due to the fact that I/O pad frequency is usually substantially lower than the core frequency. The logic definition of the behavior of the EE pins outputs assumes that the frequency of the I/O pad is no less than 4 times slower than the core frequency. In order to support this, a toggle cannot occur in two consecutive cycles. A toggle can occur in a cycle only if there was no toggle in the preceding cycle. If the frequency of the core is more than 4 times the I/O pad maximum supported frequency, then not all event sequences could be properly reflected on the pads, and some extra logic may be required between the EOnCE EE pin outputs and the I/O pads to buffer or compress events.

4.7.6.1.2 Detecting Entry into Debug State

The EE1 signal can be configured as an indication of debug state. Each time the core enters debug state, the EE1 signal is asserted. On exiting debug state, the EE1 signal is negated. This technique can be used as a debug acknowledge.

4.7.6.1.3 Status Bit of the ERVC Register

The EE3 signal can be programmed to serve as an indication that the ERVC register (read by the core) is empty. This capability provides interrupt driven transfers to the host debugger. If the EE3 signal is programmed in this way, it is asserted when the host has finished writing to the ERVC register through the JTAG. It is negated when the core finishes reading the Most Significant Part of the ERVC register.

4.7.6.1.4 Status Bit of the ETRSMT Register

The EE4 signal can be programmed to serve as an indication of data availability in the ETRSMT register. This capability provides interrupt driven transfers to the host debugger. If the EE4 signal is programmed in this way, each time the core performs the transfer (and writes to the ETRSMT register), the EE4 signal is asserted and the host is interrupted. The EE4 signal is negated when the host has finished reading the ETRSMT register through the JTAG.

4.7.6.2 EE Signals as Inputs

EE signals can be programmed to enable event detection channels or to generate one of the EOnCE events. After reset, the EE signals are set as inputs. When programmed as an input, an EE signal must be driven with zero or one. EE assertion can be programmed to perform several functions. For example, EE2 can enable both EDCA2 and the event counter as well as generate any of the EOnCE events at the same time.

4.7.6.2.1 Using EE Signals to Enable Event Detection Channels

Each EE signal can be programmed to enable the corresponding address detection channel or the data detection channel. The user can configure EE0 to enable EDCA0, EE1 to enable EDCA1, EE2 to enable EDCA2, and so on. EED can also be configured to enable EDCD. For a description of how address event detection channels can be configured to be enabled upon an appropriate EE assertion, see [Section 4.9.1.1, “EDCA Control Registers \(EDCAi_CTRL\),”](#) and [Section 4.9.2.1, “EDCD Control Register \(EDCD_CTRL\).”](#)

4.7.6.2.2 Using EE Signals to Cause EOnCE Events

If programmed by the user, EE signal assertion can cause any of the following EOnCE events:

- Place the core in debug state.
- Cause a debug exception.
- Enable trace buffer.
- Disable trace buffer.

4.7.6.2.3 Using EE Signals to Enter Debug State

The EE0 signal by default can cause the core to enter debug state right after core reset. It can also cause the core to leave a wait or stop state and enter debug state.

4.7.6.3 EE Signals Control Register (EE_CTRL)

This 16-bit register defines the behavior of the EE signals.

Figure 4-18 displays the bit configuration of the EE signals control register. Shaded bits are reserved and should be initialized with zeros for future software compatibility.

	BIT 15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	BIT 0
	EEDDEF					EE5DEF	EE4DEF	EE3DEF	EE2DEF	EE1DEF	EE0DEF					
TYPE	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw
RESET	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1

Figure 4-18. EE Signals Control Register (EE_CTRL)

The functionality of EE signals when programmed as an input depends on the programming of the EDU and the ES. See [Section 4.9, “Event Detection Unit \(EDU\) Channels and Registers,”](#) for further details.

Table 4-16 describes the EE_CTRL fields.

Table 4-16. EE_CTRL Description

Name	Description	Settings
EEDEF Bit 15	EE Definition — Programs the EE signal. As an output of the EED, the EEDEF bit can indicate detection by the EDCA, working as a toggle. As an input to the EOnCE, EE can be programmed to enable the EDCA. EE cannot disable EDCA.	0 = Output, detection by EDCA 1 = Input, enables EDCA
R Bits 14–11	Reserved	
EE5DEF Bit 10	EE5 Definition — Programs the EE5 signal. Programmed as an output of the EOnCE, EE5 can indicate detection by EDCA5, working as a toggle. Programmed as an input to the EOnCE, EE5 can be programmed to enable EDCA5. EE5 cannot disable EDCA5.	0 = Output, detection by EDCA5 1 = Input, enables EDCA5
EE4DEF Bits 9–8	EE4 Definition — Programs the EE4 signal. Programmed as an output of the EOnCE, EE4 can indicate detection by EDCA4, working as a toggle. It can also indicate that the ETRSM register was written by the core. Programmed as an input to the EOnCE according to the programming of the EDU and the ES, EE4 can be programmed to enable EDCA4 or to generate one of the EOnCE events. EE4 cannot disable EDCA4.	00 = Output, detection by EDCA4 01 = Output, data in ETRSM register ready 10 = Reserved 11 = Input, enables EDCA4 or generates an EOnCE event
EE3DEF Bits 7–6	EE3 Definition — Programs the EE3 signal. Programmed as an output of the EOnCE, EE3 can indicate detection by EDCA3, working as a toggle. It can also indicate that the ERV register is full. Programmed as an input to the EOnCE according to the programming of the EDU and the ES, EE3 can be programmed to enable EDCA3 or to generate one of the EOnCE events. EE3 cannot disable EDCA3.	00 = Output, detection by EDCA3 01 = ERV register full 10 = Reserved 11 = Input, enables EDCA3 or generates an EOnCE event

Table 4-16. EE_CTRL Description (Continued)

Name	Description	Settings
EE2DEF Bits 5–4	<p>EE2 Definition — Programs the EE2 signal. Programmed as an output of the EOnCE, EE2 can indicate detection by EDCA2, working as a toggle.</p> <p>Programmed as an input to the EOnCE according to the programming of the ECNT, EDU, and ES, EE2 can be programmed to:</p> <ul style="list-style-type: none"> • Enable the ECNT together with the ECNTEN bits • Enable the EDCA2 together with the EDCAEN bits • Generate one of the EOnCE events together with the ES <p>EE2 cannot disable EDCA2 or ECNT.</p>	00 = Output, detection by EDCA2 01 = Reserved 10 = Reserved 11 = Input, enables EDCA2 or ECNT, or generates an EOnCE event
EE1DEF Bits 3–2	<p>EE1 Definition — Programs the EE1 signal. Programmed as an output of the EOnCE, EE1 can indicate detection by EDCA1, working as a toggle. It can also indicate that the core has entered debug state (debug acknowledge). In the case of debug acknowledge, when single-stepping, EE1 does not toggle.</p> <p>Programmed as an input to the EOnCE according to the programming of the EDU and the ES, EE1 can be programmed to enable EDCA1 or to generate one of the EOnCE events.</p>	00 = Output, detection by EDCA1 01 = Debug acknowledgement 10 = Reserved 11 = Input, enables EDCA1 or generates an EOnCE event
EE0DEF Bits 1–0	<p>EE0 Definition — Programs the EE0 signal. Programmed as an output of the EOnCE, EE0 can indicate detection by the EDCA0, working as a toggle.</p> <p>Programmed as an input to the EOnCE according to the programming of the EDU and the ES, the EE0 can be programmed to enable EDCA0 together with the EDCAEN bits, or generate one of the EOnCE events together with the ES.</p> <p>EE0 can also be programmed to force the core into debug state. This default state enables entry into debug state directly after core reset. Holding EE0 at logic value 1 during and after the reset enters the core into debug state before the first dispatch occurs. In this mode, asserting EE0 also causes an exit from stop or wait processing states of the core, as specified in Section 5.7, “Processing States,” on page 5-41.</p> <p>When programmed as a debug request, EE0 can also enable EDCA0 or generate an EOnCE event if EDCA0 or the ES are programmed in this manner.</p>	00 = Output, detection by EDCA0 01 = Reserved 10 = Input, enables EDCA0 or generates an EOnCE event 11 = Input, debug request (also enables EDCA0 or generates an EOnCE event)

4.7.7 Core Command Register (CORE_CMD)

The CORE_CMD register is used to execute instructions in the core while in debug state. The external host writes the instruction into the CORE_CMD register as described in [Section 4.2.6, “Reading/Writing EOnCE Registers Through JTAG.”](#) The EOnCE commands written into the ECR must be Write CORE_CMD and GO. After writing the instruction into the CORE_CMD register, the core executes it without leaving debug state. If the EX bit in ECR is also set, debug state is exited after the instruction is performed.

The format of the injected command is shown in Figure 4-19 below.

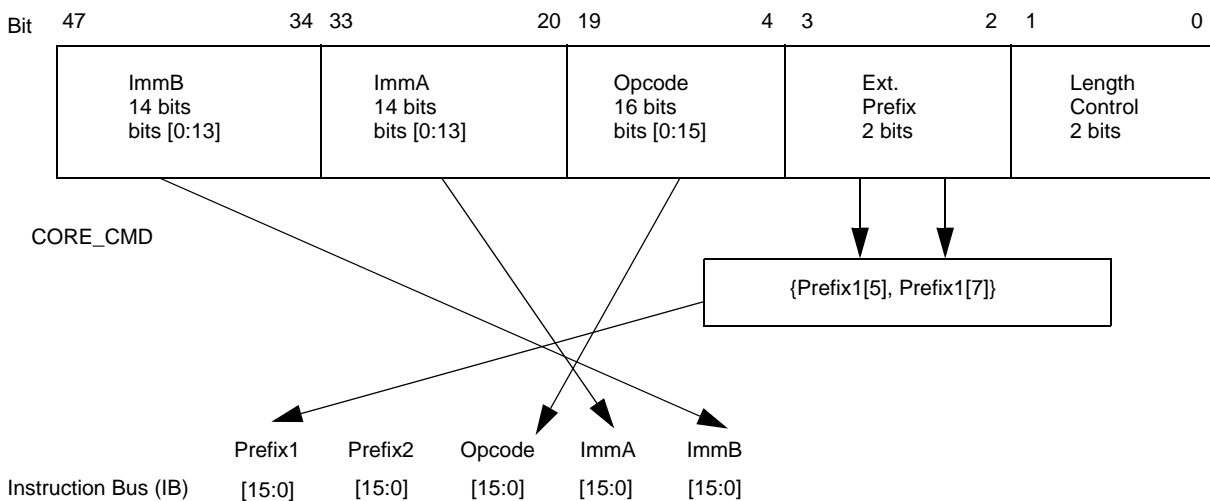


Figure 4-19. Injected Instruction Format

The length control bits are described in Table 4-17, below:

Table 4-17. Length Control Bits

Length Control Bits		Description
0	0	Not supported
0	1	One word instruction
1	0	Two word instruction
1	1	Three word instruction

The two prefix bits allow the instruction to use the high bank of registers. Bits 15 and 14 in the second and third words encode the grouping/word partition used by the core for execution set parsing. In the case of a single instruction, they do not need to be part of the CORE_CMD word. For further details, see [Appendix A, “SC140 DSP Core Instruction Set.”](#)

In general, core commands should not perform illegal operations. In case a core command generated an exception (such as an illegal exception), the exception will be serviced only after the core exits debug state.



4.7.8 PC of the Exception Execution Set (PC_EXCP)

PC_EXCP enables the user to determine exactly which execution set caused an imprecise internal Illegal or DALU overflow exception. It is a read-only register that is accessed through the JTAG port or by core software.

In the case of an illegal instruction, illegal execution set or DALU overflow, the PC of the execution set is saved in the PC_EXCP register. In this way, users can determine the address of the execution set that caused the internal exception. It is best done in the illegal exception service routine, which is serviced right after the exception event has occurred. For a list of internal exceptions, see [Section 5.8, “Exception Processing,”](#) on page 5-46.

Multiple exception events may occur between the first event and its exception service routine. For multiple exception types (illegal or DALU overflow), the PC_EXCP register will capture the VLES address of the first occurrence of the last exception type, regardless of whether the exception type is serviced or not. For multiple events of the same type (including the different reasons of the illegal exception), only the first event will be sampled in PC_EXCP.

4.7.9 PC of the Next Execution Set (PC_NEXT)

PC_NEXT is a 32-bit register that stores the address of the execution set to be executed next. Although the PC_NEXT register can also be read while the device is running and not in debug state, the register contents are not defined. This register is not affected by the operations performed during debug state. When single stepping, the value of PC_NEXT is valid after every step. PC_NEXT will sample data only if the EOnCE is enabled (see [Section 4.4, “EOnCE Enabling and Power Considerations.”](#)) If the EOnCE enters debug state without being enabled first, the value of PC_NEXT is undefined.

PC_NEXT is read-only and read through JTAG.

4.7.10 PC of Last Execution Set (PC_LAST)

PC_LAST contains the PC of the last executed execution set. It is used in debug state to define which PC triggered a PC breakpoint. If the PC_LAST register is read while the device is running and not in debug state, the register contents are not defined. PC_LAST will sample data only if the EOnCE is enabled (see [Section 4.4, “EOnCE Enabling and Power Considerations.”](#)) If the EOnCE enters debug state without being enabled first, the value of PC_LAST is undefined. In case of a killed PC (when PCKILL in ESR is asserted), the value of PC_LAST is not updated to that of the killed PC, and still reflects the last execution set.

PC_LAST is read-only and read through JTAG.

4.7.11 PC Breakpoint Detection Register (PC_DETECT)

PC_DETECT captures the PC value of the first execution set that caused an entry into debug state based on a data memory event in EDCA or EDCD. For data breakpoint detection, only the first event will be sampled into PC_DETECT because the core has already executed a few more execution sets by the time it enters debug state.

PC_DETECT captures the correct PC of the VLES that triggered the entry into debug state if all the following conditions are met:

- An event was detected on XABA/B and/or XDBA/B by an enabled EDCA/D channel.

- This event was programmed in ESEL_DM.
- The debug reason bits (DREDCA0-5, DREDCD) in ESR indicate that the data detection event was part of the reason to enter debug state. ESR should be checked because ESEL_DM may be programmed to enter debug state for other reasons that do not cause sampling into PC_DETECT.

This ESR check should match the way the data memory events were programmed to combine to a debug entry condition in ESEL_CTRL. For example, several conditions could be ANDed or ORed, requiring a different ESR check in each case.

Once sampled, PC_DETECT will not be re-sampled again until the core enters and then exits debug state.

PC_DETECT is read-only and read through JTAG or by core software. PC_DETECT should be read only in debug state. If PC_DETECT is read when the core is not in debug state, its value is undefined.

4.8 Event Counter Registers

The event counter (ECNT) contains three registers:

- Event Counter Register (ECNT_CTRL)
- Event Counter Value Register (ECNT_VAL)
- Extension Counter Value Register (ECNT_EXT)

These three registers are described in the following sections.

4.8.1 Event Counter Control Register (ECNT_CTRL)

The ECNT_CTRL register selects the event to be counted by the event counter. It also determines the enabled source of the event counter.

Two modes of event counter operation are determined by the ECNT_CTRL register:

1. In the regular mode of operation, the extension counter is disabled. Thus, when the event counter reaches zero, the count event is generated and the counter stops its operation. The maximum value that can be counted before generating the count event is \$8000 0000. This can be achieved by writing \$0000 0000 to the ECNT_VAL register. The event counter can be used as a watchdog timer provided that the counter is programmed to count the DSP cycles (internal clock), and that the debug exception in the ES event is set to generate an EOnCE event upon count event (when the counter comes to zero).
2. In the extended mode of operation, when the event counter reaches zero, it does not generate the count event and wraps around to \$7FFF FFFF. The event counter continues to count down, and the number of transitions from 1 to 0 is counted by the extension counter. This creates a virtual 62-bit counter. When the extension counter (ECNT_VAL) reaches \$7FFF FFFF, the next count wraps around to \$0000 0000. Overflow of the extension counter register does not generate a count event.

For information on events 0–5, see [Section 4.9.1, “Address Event Detection Channel \(EDCA\).”](#) For information on event D, see [Section 4.9.2, “Data Event Detection Channel \(EDCD\).”](#) Like EDU, the event counter can be enabled explicitly by writing 1111 to the ECNTEN bits of the control register. It can also be enabled by specifying an event. The profiler can exploit this capability for cycle count operations to

ascertain the number of cycles needed by a device to get from a starting address to an ending address, in the following manner:

1. Write \$7FFF FFFF to the ECNT_VAL register.
2. Configure ECNT to count the internal clock.
3. Program ECNT to be enabled upon EDCA_i detection.
4. Program EDCA_i to detect the starting address.
5. Program EDCA_j to detect the ending address.
6. Program ES to generate a debug exception upon EDCA_j detection.

The following stages are:

1. Detection of the start address which enables the counter and to start counting.
2. Detection of the final address which generates a debug exception.
3. ISR of the debug exception which disables the counter, reads the counter contents (ECNT_VAL register), and subtracts the cycles of the interrupt service routine overhead. This value gives the cycle count between the count enabling and the ending address.

When the trace buffer operates in counter mode, each destination address that is put into the trace buffer is followed by the value of the counter register. The trace buffer can also be configured to write both the values of the counter and extension counter with each trace package. For more information, see [Section 4.11.1, “Trace Buffer Control Register \(TB_CTRL\).”](#)

The counter can be configured to count the number of traced entries. In case the tracing includes the counter values themselves, they are counted as well.

Figure 4-20 displays the configuration of ECNT_CTRL. The shaded bits are reserved and should be initialized with zeros for future software compatibility.

	BIT 15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	BIT 0
							TEST	EXT	ECNTEN			ECNTWHAT				
TYPE	r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w
RESET	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Figure 4-20. Event Counter Register (ECNT_CTRL)

Table 4-18 describes the ECNT_CTRL fields.

Table 4-18. ECNT_CTRL Description

Name	Description	Settings
R Bits 15–10	Reserved	
TEST Bit 9	Reserved for Test	0 = Normal operation 1 = Reserved for test
EXT Bit 8	Extended Mode of Operation Bit — See Section 4.5.2, “Event Counter,” on page 4-18.	0 = ECNT operates in regular mode 1 = ECNT operates in extended mode



Table 4-18. ECNT_CTRL Description (Continued)

Name	Description	Settings
<p>ECNTEN Bits 7–4</p>	<p>Event Counter Enable — Used to enable the ECNT operation. When ECNTEN is set to 1111, ECNT is operational and will count events according to ECNTWHAT bits, which select the source for that count. If bits ECNTEN are set to enable the operation of the event counter when an event is detected or signal EE2 is asserted, the EOnCE overwrites these bits to 1111 one cycle after the appearance of the event.</p> <p>When the event counter is programmed to be enabled by the same event that it has to count, the first such event enables the event counter and is counted as the first event.</p> <p>When the event counter is enabled by a given event, but is programmed to count a different event, the counter does not include the enabling event in the count.</p>	<p>0000 = The event is disabled. 0001 = The event counter is disabled, but is enabled when an event is detected by the EDCA0. 0010 = The event counter is disabled, but is enabled when an event is detected by the EDCA1. 0011 = The event counter is disabled, but is enabled when an event is detected by the EDCA2. 0100 = The event counter is disabled, but is enabled when an event is detected by the EDCA3. 0101 = The event counter is disabled, but is enabled when an event is detected by the EDCA4. 0110 = The event counter is disabled, but is enabled when an event is detected by the EDCA5. 0111 = The event counter is disabled, but is enabled when an event is detected by the optional external EDCA6. 1000 = The event counter is disabled, but is enabled when an event is detected by the optional external EDCA7. 1001 = The event counter is disabled, but is enabled when an event is detected by EDCD. 1010 = The event counter is disabled, but is enabled when signal EE2 is asserted and EE2 is programmed in the EE_CTRL register as an input. 1011 = Reserved 1100 = Reserved 1101 = Reserved 1110 = Reserved 1111 = The event counter is enabled.</p>
<p>ECNTWHAT Bits 3–0</p>	<p>Events to be Counted — Determines what is to be counted by ECNT.</p>	<p>0000 = Count event0 occurrence. 0001 = Count event1 occurrence. 0010 = Count event2 occurrence. 0011 = Count event3 occurrence. 0100 = Count event4 occurrence. 0101 = Count event5 occurrence. 0110 = Count optional external event6 occurrence 0111 = Count optional external event7 occurrence 1000 = Count eventD occurrence. 1001 = Count executions of DEBUGEV instruction. 1010 = Count trace events (data moved to the buffer). 1011 = Count executed execution sets. 1100 = Count core clocks. 1101 = Count off-core event 0 1110 = Count off-core event 1 1111 = Reserved</p>

4.8.2 Event Counter Value Register (ECNT_VAL)

This 32-bit register is used to determine how many events the event counter should count before it generates the count event signal. ECNT_VAL is a down-counter. The MSB is always zero, so the range is from \$7FFF FFFF to \$0000 0000. When the register is written, the MSB should be written to zero for software compatibility.



4.8.3 Extension Counter Value Register (ECNT_EXT)

This is a 32-bit register that is used in the extended mode of operation to count the number transitions from 1 to 0 in the ECNT_VAL register. See [Section 4.5.2, “Event Counter,”](#) for further details. The ECNT_EXT register counts up. Reset writes zeros to this register. Software can write the register when new counting is started. The MSB is always zero, so the count is from \$0000 0000 to \$7FFF FFFF. When the register is written, the MSB should be written to zero for software compatibility.

4.8.4 EC Signals

The two event counter signals EC0 and EC1 allow the event counter to count off-core events such as cache hits/misses, memory contention, external wait states, etc. These inputs are assumed to be synchronized to the core clock and support a counting rate up to the core frequency. EC0 and EC1 use is derivative-dependent.

4.9 Event Detection Unit (EDU) Channels and Registers

The various event detection channels and corresponding registers are described in the sections that follow.

4.9.1 Address Event Detection Channel (EDCA)

The EDCA can be used to detect the following:

- Program breakpoint, specified on a specific PC value or range
- Data address breakpoint, specified on a specific data address or range.

The functionality of these registers is described in the following sections.

4.9.1.1 EDCA Control Registers (EDCA_i_CTRL)

EDCA_i_CTRL is a 16-bit register used to control the behavior of the corresponding EDCA. The following sections describe the functionality of each bit in the EDCA_i_CTRL register.

Figure 4-21 displays the configuration of the EDCA_i_CTRL register.

	BIT 15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	BIT 0
			EDCAEN				CS		CBCS		CACCS		ATS		BS	
TYPE	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw
RESET	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Figure 4-21. EDCA Control Register (EDCA_i_CTRL)

Table 4-19 describes the EDCA_i_CTRL fields.

Table 4-19. EDCA_CTRL Description

Name	Description	Settings
R Bits 14-15	Reserved	

Table 4-19. EDCA_CTRL Description (Continued)

Name	Description	Settings
EDCAEN Bits 13–10	<p>Event Detection Channel (EDCA_{<i>i</i>}) Enable — Used to enable or disable event detection channels. When it is enabled, it continues to operate until it is explicitly disabled by writing 0000 into EDCAEN bits, or EDCAEN bits are changed for another enabling condition. The channel remains disabled until a new enabling condition occurs.</p> <p>When bits EDCAEN are set to enable the operation of EDCA_{<i>i</i>} upon detection of an event, the EOnCE overwrites these bits to 1111 one cycle after the appearance of the event.</p> <p>When the event detection channel is programmed to detect PC, the PC of the first instruction cycle after the channel has been enabled is not yet detected.</p> <p>The latency for enabling a channel is one cycle.</p>	0000 = EDCA _{<i>i</i>} is disabled. 0001 = EDCA _{<i>i</i>} is disabled, but is enabled when an event is detected by EDCA0. 0010 = EDCA _{<i>i</i>} is disabled, but is enabled when an event is detected by EDCA1. 0011 = EDCA _{<i>i</i>} is disabled, but is enabled when an event is detected by EDCA2. 0100 = EDCA _{<i>i</i>} is disabled, but is enabled when an event is detected by EDCA3. 0101 = EDCA _{<i>i</i>} is disabled, but is enabled when an event is detected by EDCA4. 0110 = EDCA _{<i>i</i>} is disabled, but is enabled when an event is detected by EDCA5. 0111 = EDCA _{<i>i</i>} is disabled, but is enabled when an event is detected by the optional external EDCA6 1000 = EDCA _{<i>i</i>} is disabled, but is enabled when an event is detected by the optional external EDCA7. 1001 = EDCA _{<i>i</i>} is disabled, but is enabled when an event is detected by EDCD. 1010 = EDCA _{<i>i</i>} is disabled, but is enabled when a count event is detected. 1011 = EDCA _{<i>i</i>} is disabled, but is enabled when a signal EE _{<i>i</i>} is asserted and EE _{<i>i</i>} in the EE_CTRL register is programmed to be an input. 1100 = EDCA _{<i>i</i>} is enabled but will be disabled when EE _{<i>i</i>} is negated, in both cases and EE _{<i>i</i>} is programmed as an input in the EE_CTRL register. This state can only be reached by previously being in the 1011 state and asserting the EE _{<i>i</i>} pin. 1101 = Reserved 1110 = Reserved 1111 = EDCA is enabled.
CS Bits 9–8	<p>Comparators Selection — Used to select the desired combination of comparator A and comparator B results. An event detection can be generated in the following cases:</p> <ul style="list-style-type: none"> • Only comparator A condition is detected. • Only comparator B condition is detected. • Both comparator A and comparator B conditions are detected. • Either comparator A or comparator B conditions are detected. 	00 = Comparator A only 01 = Comparator B only 10 = Comparator A AND Comparator B 11 = Comparator A OR Comparator B

Table 4-19. EDCA_CTRL Description (Continued)

Name	Description	Settings
CBCS Bits 7–6	Comparator B Condition Selection — Used to select one of these four results from comparator B: <ul style="list-style-type: none"> • Equal to • Not equal to • Greater than • Less than 	00 = Equal to EDCA_REFB 01 = Not equal to EDCA_REFB 10 = Greater than EDCA_REFB 11 = Less than EDCA_REFB
CACS Bits 5–4	Comparator A Condition Selection — Used to select one of these four results from comparator A: <ul style="list-style-type: none"> • Equal to • Not equal to • Greater than • Less than 	00 = Equal to EDCA_REFA 01 = Not equal to EDCA_REFA 10 = Greater than EDCA_REFA 11 = Less than EDCA_REFA
ATS Bits 3–2	Access Type Selection — These bits are used to select the type of memory access that should be detected by the event detection channel. The possible memory access types are: <ul style="list-style-type: none"> • Read access • Write access • Read or write access 	00 = Read access 01 = Write access 10 = Read or write access 11 = Reserved
BS Bits 1–0	Bus Selection — Used to select which address bus or buses should be sampled for comparison by comparator A and/or by comparator B. The possible buses that can be chosen by these bits are PC, XABA, and XABB. If XABA and/or XABB are the selected buses, bus XABA is compared to the EDCA_REFA register while XABB is compared to the EDCA_REFB register. If CS bits are set to 11, then setting BS bits to 10 enables the user to set a watchpoint on an address when it is not known whether the address to be used for accessing memory is on XABA or XABB. In this case, the user must set both reference value registers with the address to be detected. If the BS bits are set to 10, then the CS bits must be set to either 10 or 11.	00 = XABA 01 = XABB 10 = XABA and XABB 11 = PC

In order to detect a watchpoint on a PC range, one EDCA is enough. In order to detect a watchpoint on a data address range, two EDCAs are required. When configuring two EDCAs to detect a watchpoint to an address range, one EDCA should be configured to detect the range on bus A, and the other EDCA to detect



the range on bus B, and the two EDCA events should be OR-ed in the event selector. The apparent alternative of detecting the upper range boundary on both buses and the other EDCA to detect the lower range boundary on both buses (AND-ing them in the event selector) may produce erroneous results, for example if two unrelated parallel accesses match the conditions by chance.

When used for data address detection, the EDCA takes into account the access width in order to identify an access to an address that is part of a wide access. Hence, when the data access is 16-bit wide, the LSB of the address does not participate in the comparison.

In a similar way, the following holds true:

- for 16-bit accesses - the LSB of the address is not compared
- for 32-bit accesses (including MOVE.2W) - the 2 LSBs of the address are not compared
- for 64-bit accesses (including MOVE.4W, MOVE.2W) - the 3 LSBs of the address are not compared.

For example, this feature allows setting a watchpoint on address 0x101, in which case the EDCA will also identify it when the core is performing a MOVE.W, MOVE.L, etc. to address \$100.

When detection occurs, status bit $EDCAST_i$ is set by the EOnCE in the EMCR register. Refer to Table 4-15 on page 4-41.

4.9.1.2 EDCA Reference Value Registers A and B ($EDCA_i_REFA$, $EDCA_i_REFB$)

$EDCA_i_REFA$ and $EDCA_i_REFB$ are 32-bit registers used to hold reference values that are to be compared by the event detection channel comparators. $EDCA_i_REFA$ is used by the event detection channel comparator A. $EDCA_i_REFB$ is used by the event detection channel comparator B.

4.9.1.3 EDCA Mask Register ($EDCA_i_MASK$)

The $EDCA_i_MASK$ is a 32-bit register that allows masking of any one of a sample address' bits. The sampled address is ANDed with the mask value.

- Mask bits with a value of one stored in them allow the corresponding bit of the selected address to participate in the comparison.
- Mask bits with a value of zero stored in them cause the corresponding bit of the selected address to always match the corresponding bits in the reference value.

The masked address value is then compared to the $EDCA_i_REFA$ and $EDCA_i_REFB$ registers. For example, the $EDCA_i_MASK$ register can be used to detect accesses to a memory region with several address aliases.

4.9.2 Data Event Detection Channel (EDCD)

In order to set a watchpoint on a given data value, the user should:

- Write the watched value into the EDCD_REF.
- Enter a write mask into the EDCD_MASK.
- Specify the type of access (read or write) in the EDCD_CTRL.
- Specify the data type (byte/word/long) in the EDCD_CTRL.
- Enable the event detection unit in EDCD_CTRL, as the last action.

The following sections describe the functionality of these registers.

4.9.2.1 EDCD Control Register (EDCD_CTRL)

Figure 4-22 displays the configuration of EDCD_CTRL.

	BIT 15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	BIT 0		
									AWS					EDCDEN		CCS		ATS
TYPE	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	
RESET	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	

The shaded bits are reserved and should be initialized with zeros for future software compatibility.

Figure 4-22. EDCD Control Register (EDCD_CTRL)

Table 4-20 describes the EDCD_CTRL fields.

Table 4-20. EDCD_CTRL Description

Name	Description	Settings
R Bits 14-15	Reserved	
R Bits 13-10	Reserved	

Table 4-20. EDCD_CTRL Description (Continued)

Name	Description	Settings
AWS Bits 9–8	<p>Access Width Selection — Determines the width of the data access that should be watched. The different width types are summarized in the settings column.</p> <p>In byte access mode, only the eight LSB bits of the masked data are compared with the eight LSB bits of the EDCD_REF register. One or four comparisons are performed with logical OR among them, depending on the access resolution (byte or quad-byte).</p> <p>When word access width is chosen, the sixteen LSB bits of the EDCD_REF register are compared with each of the sixteen bits of the masked data. One, two, or four comparisons are performed with logical OR among them, depending on the access resolution (word, double-word, or quad-word).</p> <p>When long access width is chosen, then the 32 LSB bits of the EDCD_REF register are compared with each of the 32 bits of the masked data. One or two comparisons are performed with logical OR among them, depending on the access resolution (long or double-long).</p>	00 = Byte 01 = Word 10 = Long 11 = Reserved
R Bit 7	Reserved	



Table 4-20. EDCD_CTRL Description (Continued)

Name	Description	Settings
<p>EDCDEN Bits 6–3</p>	<p>EDCD Enable — Used to enable or disable the EDCD. When enabled, EDCD continues to operate until it is explicitly disabled by writing 0000 into EDCDEN bits, or when EDCDEN bits are changed for another enabling condition. The channel remains disabled until a new enabling condition occurs.</p> <p>When the EDCDEN bits are set to enable the operation of the EDCD upon event occurrence, the EOnCE overwrites these bits to 1111 one clock cycle after the appearance of the event. The latency for enabling the channel is one cycle.</p>	<p>0000 = EDCD is disabled. 0001 = EDCD is disabled, but is enabled when an event is detected by EDCA0. 0010 = EDCD is disabled, but is enabled when an event is detected by EDCA1. 0011 = EDCD is disabled, but is enabled when an event is detected by EDCA2. 0100 = EDCD is disabled, but is enabled when an event is detected by EDCA3. 0101 = EDCD is disabled, but is enabled when an event is detected by EDCA4. 0110 = EDCD is disabled, but is enabled when an event is detected by EDCA5. 0111 = EDCD is disabled, but is enabled when an event is detected by the optional external EDCA6. 1000 = EDCD is disabled, but is enabled when an event is detected by the optional external EDCA7. 1001 = EDCD is disabled, but is enabled when a count event is detected. 1010 = EDCD is disabled, but is enabled when EED is asserted and EED is programmed as input in the EE_CTRL register. 1011 = EDCD is enabled but will be disabled when EED is negated, in both cases EED is programmed as an input in the EE_CTRL register. This state can only be reached by previously being in the 1010 state and asserting the EED pin. 1100 = Reserved 1101 = Reserved 1110 = Reserved 1111 = EDCD is enabled.</p>
<p>CCS Bits 2–1</p>	<p>Comparator Condition Selection — These bits select one of these four results from the comparator:</p> <ul style="list-style-type: none"> • Equal to • Not equal to • Greater than • Less than 	<p>00 = Equal to EDCD_REF 01 = Not equal to EDCD_REF 10 = Greater than EDCD_REF 11 = Less than EDCD_REF</p> <p>In case of multi-operand data accesses (such as MOVE.2W etc.) the compare result will be true if the condition is fulfilled for any of the individual operands (one byte, word etc.). However for the “not equal” condition – <i>all</i> operands must be not equal in order for the condition to be fulfilled.</p>
<p>ATS Bit 0</p>	<p>Access Type Selection — The ATS bit determines whether the memory access is read or write.</p>	<p>0 = Read 1 = Write</p>

4.9.2.2 EDCD Reference Value Register (EDCD_REF)

EDCD_REF is a 32-bit register used to hold a reference value to be compared by the EDCD comparator. EDCD_REF is used by the EDCD comparator. If a byte (8 bits) or a word (16 bits) is to be written into the EDCD_REF, it should be LSB-aligned.

4.9.2.3 EDCD Mask Register (EDCD_MASK)

EDCD_MASK is a 32-bit register that allows the masking of any of the bits in the data bus value that is compared. If bit i in the EDCD_MASK is zero, then it does not participate in the comparison. The data on the bus is ANDed with the EDCD_MASK value, and is then compared with the EDCD_REF register according to the access width. Hence the masked bits must also be zero in the EDCD_REF.

In case that the reference value is 16 or 8 bits long, the mask value must be duplicated 2 or 4 times (respectively) to fill the mask register.

Since data buses are 64 bits, the 32-bit EDCD_MASK register is masked on both 32 LSB and 32 MSB bits of the sampled bus value.

EDCD_MASK is initialized to all ones at reset.

4.10 Event Selector (ES) Registers

ES selects the source for various operations used by the EOnCE. It contains the following registers:

- Event selector control register (ESEL_CTRL)
- Event selector mask debug state register (ESEL_DM)
- Event selector mask debug exception register (ESEL_DI)
- Event selector mask enable trace register (ESEL_ETB)
- Event selector mask disable trace register (ESEL_DTB)

The following sections describe these registers.

4.10.1 Event Selector Control Register (ESEL_CTRL)

The 8-bit control register ESEL_CTRL controls the operation of the ES, which is programmed in the following order:

1. Reset the event selector mask registers.
2. Program the ESEL_CTRL register.
3. Program the appropriate event selector mask registers.

Figure 4-23 displays the bit configuration of ESEL_CTRL.

	BIT 7	6	5	4	3	2	1	BIT 0
				SELDTB	SELETB		SELDI	SELDM
TYPE	rw	rw	rw	rw	rw	rw	rw	rw
RESET	0	0	0	0	0	0	0	0

The shaded bits are reserved and should be initialized with zeros for future software compatibility.

Figure 4-23. Event Selector Control Register (ESEL_CTRL)

The ESEL_CTRL fields are described in Table 4-21.

Table 4-21. ESEL_CTRL Description

Name	Description	Settings
R Bits 7–5	Reserved	
SELDTB Bit 4	Selection Bit for Trace Disable — Determines how the enabled sources disable trace.	0 = Trace is disabled upon detection of the event by any one of the sources (ORed) selected on the ESEL_DTB register. 1 = Trace is disabled upon detection of the event by all the sources (ANDed) selected on the ESEL_DTB register.
SELETB Bit 3	Selection Bit for Trace Enable — Determines how the enabled sources enable trace.	0 = Trace is enabled upon detection of the event by any one of the sources (ORed) selected on the ESEL_ETB register. 1 = Trace is enabled upon detection of the event by all the sources (ANDed) selected on the ESEL_ETB register.
R Bit 2	Reserved	
SELDI Bit 1	Selection Bit for Debug Exception — Determines how the enabled sources cause a debug exception.	0 = A debug exception is reached upon detection of the event by any one of the sources (ORed) selected on the ESEL_DI register. 1 = A debug exception is reached upon detection of the event by all the sources (ANDed) selected on the ESEL_DI register.
SELDM Bit 0	Selection Bit for Debug state — Determines how the enabled sources cause the core to enter into debug state.	0 = Core enters debug state upon detection of the event by any one of the sources (ORed) selected on the ESEL_DM register. 1 = Core enters debug state upon detection of the event by all the sources (ANDed) selected on the ESEL_DM register.

Each of the following event selector registers can enable the system to configure what debug events (EDCA event, EE event etc.) will cause the outcome controlled by that register (entry into debug state, debug exception etc.).



For each outcome, the individual events could be AND-ed or OR-ed as specified in ESEL_CTRL. When the ESEL_CTRL is configured as “OR” for a certain outcome, all events could be enabled as needed. However, AND-ing events is more restricted, since the respective events that are AND-ed have to be related so that the combination will be meaningful. For example, there is no meaning to AND an EE pin assertion (a transient event) with an EDCA event because there is no practical way to synchronize them.

Combining events with an AND condition in the ESEL_CTRL register is allowed only for the following options:

- EDCA0-5 events all configured to detect a PC meaning: the events relate to the same execution set
- One EDCA0-5 event configured to detect data accesses with an EDCD event meaning: a matching address-data pair of the same access.

The interface for the optional external EDCA6 and EDCA7 has the same timing as data address events. This means that if the external EDCAs were implemented with the same detection timing as the internal ones, they could be used in the event selector with other events in same way as the internal EDCAs.

In general, the options for combining external EDCA events in the event selector are as follows:

- OR-ed with other EOnCE events, as asynchronous events with respect to core activity.
- AND-ed with other EOnCE events, if used as an enabling event that is asserted continuously for extended periods. In this case the assertion and de-assertion transitions of EDCA6/7 events should be controlled by the system (in SW or HW) since the transition edge cases could be problematic to detect.
- AND-ed with an EDCD event, intended to detect an address/data pair on the same bus.

In case the event selector is configured to have some events cause entry into debug state and other events to cause a debug exception (ESEL_DM and ESEL_DI are both enabled at the same time), then the events configured in each of them cannot be with an AND condition in ESEL_CTRL.

4.10.2 Event Selector Mask Debug State Register (ESEL_DM)

This 16-bit register has one bit for each source of event selection. Setting the appropriate bit configures the related source to cause entry into debug state.

Figure 4-24 displays the bit configuration of ESEL_DM.

	BIT 15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	BIT 0
	DEBUGE V	EE4	EE3	EE2	EE1	EE0	COUN T	EDCD	EDCA 7	EDCA 6	EDCA 5	EDCA 4	EDCA 3	EDCA 2	EDCA 1	EDCA 0
TYPE	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw
RESET	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Figure 4-24. Event Selector Mask Debug State (ESEL_DM)

If multiple sources are configured to cause entry into debug state, they are ANDed or ORed according to the value of the SELDM bit in the ESEL_CTRL. For more information, see [Section 4.10.1, “Event Selector Control Register \(ESEL_CTRL\),”](#) on page 4-61. If all the bits are set to zero, the ES does not enter debug state.



4.10.3 Event Selector Mask Debug Exception Register (ESEL_DI)

This 16-bit register has one bit for each source of event selection. Setting the appropriate bit enables the related source to cause a debug exception.

Figure 4-25 displays the bit configuration of ESEL_DI.

	BIT 15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	BIT 0
	DEBUGE V	EE4	EE3	EE2	EE1	EE0	COUNT	EDCD	EDCA 7	EDCA 6	EDCA 5	EDCA 4	EDCA 3	EDCA 2	EDCA 1	EDCA 0
TYPE	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw
RESET	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Figure 4-25. Event Selector Mask Debug Exception (ESEL_DI)

If multiple sources are configured to cause a debug exception, they are ANDed or ORed according to the value of the SELDI bit in the ESEL_CTRL. For more information, see [Section 4.10.1, “Event Selector Control Register \(ESEL_CTRL\).”](#) If all the bits are set to zero, the ES does not issue a debug exception.

4.10.4 Event Selector Mask Enable Trace Register (ESEL_ETB)

This 16-bit register has one bit for every source of the ES. Setting the appropriate bit configures the related source to enable trace.

Figure 4-26 displays the bit configuration of ESEL_ETB.

	BIT 15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	BIT 0
	DEBUGE V	EE4	EE3	EE2	EE1	EE0	COUN T	EDCD	EDCA 7	EDCA 6	EDCA 5	EDCA 4	EDCA 3	EDCA 2	EDCA 1	EDCA 0
TYPE	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw
RESET	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Figure 4-26. Event Selector Mask Enable Trace (ESEL_ETB)

If multiple sources are configured to enable trace, they are ANDed or ORed according to the value of the SELETB bit in the ESEL_CTRL. If all the bits are set to zero, the ES does not enable trace.

The same event cannot be configured to both enable and disable tracing.



4.10.5 Event Selector Mask Disable Trace Register (ESEL_DT B)

This 16-bit register has one bit for every source of the ES. Setting the appropriate bit configures the related source to cause a disable trace.

Figure 4-27 displays the bit configuration of ESEL_DT B.

	BIT 15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	BIT 0
	DEBUG V	EE4	EE3	EE2	EE1	EE0	COUN T	EDCD	EDCA 7	EDCA 6	EDCA 5	EDCA 4	EDCA 3	EDCA 2	EDCA 1	EDCA 0
TYPE	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw
RESET	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Figure 4-27. Event Selector Mask Disable Trace (ESEL_DT B)

If multiple sources are configured to disable trace, they are ANDed or ORed according to the value of the SELDTB bit in the ESEL_CTRL. See [Section 4.10.1, “Event Selector Control Register \(ESEL_CTRL\),”](#) for further details. If all the bits are set to zero, the ES does not issue a disable trace.

The same event cannot be configured to both enable and disable tracing.

4.11 Trace Unit Registers

The trace unit includes the following registers:

- Trace Buffer Control Register (TB_CTRL)
- Trace Buffer Read Pointer Register (TB_RD)
- Trace Buffer Write Pointer Register (TB_WR)
- Trace Buffer Virtual Register (TB_BUFF)

4.11.1 Trace Buffer Control Register (TB_CTRL)

The TB_CTRL register controls the operation of the trace unit. The following tracing modes are possible, all which trace the PC of execution sets that answer some conditions:

- TEXEXT - trace the PC of every execution set
- TMARK - trace the PC of execution sets that includes the MARK instruction
- TCHOF - trace the source and destination PC of execution set that includes a taken COF instruction (listed in Table A-13 in Appendix A, not including TRAP, but including the BREAK, CONT/D instructions)
- TLOOP - trace the execution of HW loops.
For long loops, the PC of the last address (LA) and start address (SA) are traced.
For short loops, only the PC of LA is traced.
- TINT - trace the interrupt point and destination PC of interrupts and exceptions (including the TRAP, and ILLEGAL instructions)

TEXEC and TMARK can only be activated on their own, without other tracing options enabled.

In addition, the counter values could be added to the trace package of each trace event, thereby allowing to monitor the elapsed cycles between trace events. The possible counter tracing modes are:

- **TCOUNT** upon a trace event, trace the counter value (ECNT_VAL)
- **TCNTEXT** upon a trace event, trace the extension counter value (ECNT_EXT).
This mode is useful only with the TCOUNT mode.

Activating the TCOUNT mode adds a 32-bit entry to the traced package upon each trace event. Activating the TCNTEXT mode in addition to the TCOUNT mode adds two 32-bit entries to the trace package upon each trace event.

Both the counter modes cannot be activated with the TEXEC and TMARK mode, in order not to create situations of tracing overflow.

The tracing mode combinations that are allowed are summarized in Table 4-22.

Table 4-22. Allowed tracing mode combinations

Allowed with Trace mode	TCHOF	TLOOP	TINT	TEXEC	TMARK			TCOUNT or TCOUNT & TCNTEXT
TCHOF		+	+	-	-			+
TLOOP	+		+	-	-			+
TINT	+	+		-	-			+
TEXEC	-	-	-		-			-
TMARK	-	-	-	-				-

When the TB_CTRL register is configured for multiple trace data writes, there is a potential for data loss. This is because each write to the trace buffer requires one core clock cycle. Requesting multiple trace buffer actions such as setting TLOOP, TCOUNT and TCNEXT in TB_CTRL will require a core clock cycle for each write - in this case, four clocks. If a long loop with only three execution sets is encountered with the above TB_CTRL configuration, there are not enough cycles to write all the data. The value of the extension counter register will be lost.

In TB_CTRL configurations where there are not enough core clock cycles to write all requested trace data, the priority for writing data is as follows:

1. Destination and source address
2. Counter value
3. Extension counter value

When using the supported modes, source or destination addresses could not be lost.

When tracing in all modes except TMARK and TEXEC, the LSB of the PC of the source execution set is always 1, while the LSB of all other words in a package is 0. This allows decoding the trace buffer contents when the trace buffer is set to trace different cases, when all programmed information could not be written to the trace buffer at the same time.

In order to ensure that the LSB value of the trace data is always valid according to this convention, the values of the 31-bit event counter and event counter extension are traced shifted one position to the left, occupying positions [31:1] of the traced 32-bit value. Hence for example a counter value of 0x8 will be traced as 0x10. The debugger SW should account for this shift when interpreting trace data.

When tracing in TEXEC mode, the LSB of the trace buffer entry is written with the valid T-bit value from the SR. The value is potentially set by the previous execution set, which is the previous trace buffer entry. The T bit value in a trace entry relates to the predicated DALU instructions (IFc on DALU instructions) in the *current* execution set, and relates to the predicated AGU instructions (IFc on AGU instructions) in the *next* execution set.

The tracing modes in the TB_CTRL register can only be changed when the trace buffer is disabled. Tracing could be enabled again after no less than 5 VLES from when it has been disabled.

Figure 4-28 displays the bit configuration of TB_CTRL. Shaded areas are reserved.

	BITS 15-8	7	6	5	4	3	2	1	BIT 0
		TCNTEXT	TCOUNT	TLOOP	TEN	TMARK	TEXEC	TINT	TCHOF
TYPE		rw	rw	rw	rw	rw	rw	rw	rw
RESET		0	0	0	0	0	0	0	0

Figure 4-28. Trace Buffer Control Register (TB_CTRL)

The TB_CTRL fields are described in the following table.

Table 4-23. TB_CTRL Description

Name	Description	Settings
TCNTEXT Bit 7	Trace Buffer Extension Counter Mode — Enables a special mode of the trace unit where each destination address put into the trace buffer is followed by the value of the extension counter register.	0 = Tracing of the extension counter is disabled 1 = Tracing of the extension counter is enabled
TCOUNT Bit 6	Trace Buffer Counter Mode — Enables a special mode of the trace unit where each destination address put into the trace buffer is followed by the value of the event counter register. When both counter mode bits (TCOUNT and TCNTEXT) are set, the event counter register is first written followed by the extension counter register.	0 = Tracing of the counter is disabled 1 = Tracing of the counter is enabled

Table 4-23. TB_CTRL Description (Continued)

Name	Description	Settings
TLOOP Bit 5	Trace Loops Mode — Enables tracing the addresses of hardware loops. When the bit is set, every change of flow resulting from a loop puts the last address of loop (LA) into the trace buffer. In the case of a long loop, the start address of loop (SA) is put into the trace buffer after LA. If the loop has a number of iterations N, the LA and SA of the loop are written to the trace buffer (N-1) times. The last iteration of the loop is executed in normal flow. If LC = 0 or LC = 1, LA and SA are not written to the trace buffer.	0 = Loop tracing is disabled. 1 = Loop tracing is enabled. For long loops, tracing includes LA, SA and optional counter values. For short loops, tracing includes only the PC of the loop LA
TEN Bit 4	Trace Buffer Enable Mode — Enables tracing. The TEN bit can be set or cleared directly. It can also be set when TB is enabled by the ES_ETB. It is cleared when disabled by the ES_DTB.	0 = Tracing is disabled. 1 = Trace is enabled.
TMARK Bit 3	Trace Mark Instruction Mode — Enables the trace of MARK instruction execution.	0 = MARK instruction is not traced. 1 = PC of MARK instruction is traced.
TEXEC Bit 2	Trace Issue of Execution Sets Enable Mode — Enables tracing the addresses of every issued execution set.	0 = Execution set tracing is disabled. 1 = Execution set tracing is enabled. All other mode bits should be cleared. Tracing includes the PC of every issued execution set.
TINT Bit 1	Trace Interrupts Enable Mode — Used to enable tracing the addresses of interrupt vectors. When the bit is set, each service of an interrupt puts the address of the last executed or aborted execution set (before the interrupt) into the trace buffer as well as the address of the interrupt vector.	0 = Interrupt tracing is disabled. 1 = Interrupt tracing is enabled. Tracing includes source PC of interrupt point, the PC of the interrupt vector, and optional counter values
TCHOF Bit 0	Trace Addresses of Change-of-Flow (COF) Instructions Enable Mode — Used to enable the tracing of addresses for execution sets containing change-of-flow instructions. When the bit is set, every execution of an execution set containing change-of-flow instructions (even if the change-of-flow instruction is executed together with other instructions in the execution set) puts into the trace buffer the address of that execution set (the address of the first instruction in the execution set) and the target address of the change-of-flow instruction.	0 = Tracing of COF instructions is disabled . 1 = Tracing of COF instructions is enabled. Tracing includes source PC, destination PC and optional counter value.

4.11.2 Trace Buffer Read Pointer Register (TB_RD)

TB_RD is a 16-bit register that points to the location in the RAM buffer from which the next value is read. The register is reset when the trace buffer is enabled.

4.11.3 Trace Buffer Write Pointer Register (TB_WR)

TB_WR is a 16-bit register that points to the next location available for writing into the buffer. The register is reset when the trace buffer is enabled.

4.11.4 Trace Buffer Register (TB_BUFF)

This 32-bit register is used to read the contents of the trace buffer. For details, see [Section 4.5.5.3, “Reading the Trace Buffer \(TB_BUFF\).”](#) It is a pipeline register inside the core, not the off-core trace buffer.



Chapter 5

Program Control

This chapter describes the program control features for the SC140 including:

- Pipeline
- Instruction grouping
- Instruction timing
- Hardware loops
- Stack support
- Processing states
- Exception processing

The SC140 core, being a multiple ALU processor, has special hardware that can issue up to two AGU and four DALU instructions at the same time. When two or more instructions are being issued to two or more execution units in the same clock cycle, these instructions are defined as **grouped**. The C compiler or the assembly programmer can specify in the source code which instructions are grouped together according to the SC140 programming rules. When the assembler compiles the DSP code, it specifies in the encoding whether an instruction stands alone, or whether it is grouped with other instructions. In each clock cycle, the dispatch logic detects how many instructions are grouped. Each group of instructions issued to the execution units on a given clock cycle is called an **execution set**. Each line of eight words read from the program memory and associated with an address is called a **fetch set**.

5.1 Pipeline

This section describes how instructions are processed in the SC140 core pipeline. The SC140 pipeline consists of five stages:

- Pre-fetch stage
- Fetch stage
- Dispatch stage
- Address generation stage
- Execution stage

The first three stages are implemented in the program sequencer unit (PSEQ). The last two stages are implemented in the AGU and DALU, respectively.

To support parallel execution, the core uses a variable length execution set (VLES) architecture with a static grouping mechanism. Several instructions can be grouped together to form an execution set, which is dispatched to the execution units in parallel. The core contains four ALUs and two AAUs and thus execution set can contain up to four DALU instructions and two AGU instructions with a maximum of eight words. For many instructions, an execution set takes only one clock cycle. For a detailed description of SC140 core instruction timing, see [Section 5.3, “Instruction Timing,”](#) on page 5-14.

5.1.1 Instruction Pipeline Stages

Figure 5-1 illustrates the five instruction pipeline stages.

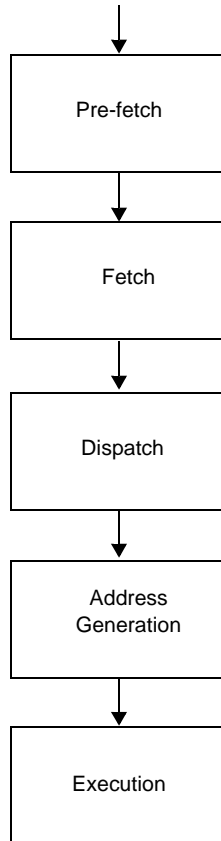


Figure 5-1. Instruction Pipeline Stages



Table 5-1 shows a typical pipeline flow. For the machine to advance to the next instruction cycle, all of the five operations at the current cycle must be completed.

Table 5-1. Pipeline Example

Operation	Instruction Cycle											
	1	2	3	4	5	6	7	8	9	10	11	12
Pre-fetch	i1	i2	i3	i4	i5	i6						
Fetch		i1	i2	i3	i4	i5	i6					
Decode			i1	i2	i3	i4	i5	i6				
Address Generation				i1	i2	i3	i4	i5	i6			
Execution					i1	i2	i3	i4	i5	i6		

Table 5-2 provides an overview of the operations performed at each stage of the pipeline.

Table 5-2. Pipeline Stages Overview

Pipeline Stage	Description
Pre-fetch	<ul style="list-style-type: none"> Generate addresses for program fetch Update fetch counter (FC)
Fetch	<ul style="list-style-type: none"> Read fetch set from memory
Dispatch	<ul style="list-style-type: none"> Dispatch instructions Decode AGU instructions
Address Generation	<ul style="list-style-type: none"> Decode DALU instructions Generate addresses for data load and store operations Perform address calculations: normal and change-of-flow Perform AGU arithmetic instructions Update AGU registers
Execution	<ul style="list-style-type: none"> Read source operands to DALU Read source register for memory store operations Perform data calculations (multiply and add) Write DALU results to destination registers Write destination register for memory load operations

5.1.1.1 Instruction Pre-Fetch and Fetch

The first two stages of the pipeline are the pre-fetch and fetch stages. These two stages combined are responsible for the program memory read of the fetch set. Each fetch set contains eight instruction words.

In the pre-fetch stage, the address of the fetch set is driven into the program address bus (PAB) along with the read strobe. This enables the memory read process. While the address is issued to memory, the fetch counter (FC) in the PSEQ is updated for the next program memory read. Both of these operations occur in parallel. The address can be generated by the PSEQ for:

- Normal program flow
- Exception program flow
- Hardware loops
- Change-of-flow instructions in the AGU

The fetch stage, which follows the pre-fetch stage, is dedicated to waiting for the memory access to be completed. Memory access is completed when the PSEQ samples the program memory value from the 128-bit wide program data bus (PDB).

Since an execution set can overlap to a second fetch set, more than one fetch set is stored in a buffer. However, the instructions in an execution set need to be dispatched together.

The SC140 core in the current implementation holds an internal fetch buffer of 4 fetch sets (128 bits each). VLES are dispatched from this buffer to the execution units, emptying the buffer. Dispatching is unrelated to fetching, and only the actual number of instruction words needed for execution are dispatched. Hence a single fetch set can be "consumed" anywhere between 1 cycle (in case the VLES is 8 words long) and 8 cycles (in case it contains 8 VLES of 1 word each). The fetch unit tries to keep this buffer full as much as it can. After reset and after every change of flow (COF), a series of 4 fetch requests is issued to the memory. Upon dispatch of the last instruction in the fetch set, another fetch request is issued to the memory. In serial code it means that a fetch is issued between once per cycle and once every 8 cycles, depending on the width of the VLES that are executed. Hence the main factors that affect the density of fetches on the program bus are the width of the execution set (number of instructions per set - affects the rate the buffer is emptied) and the frequency of COF instructions (which trigger a fetch buffer flush and re-fill).

5.1.1.2 Instruction Dispatch

After the fetch set is read from memory to the PSEQ, the PSEQ detects which instructions are grouped into an execution set. These instructions will be dispatched in parallel such that the number of cycles taken by the longest instruction will determine the number of cycles for the whole execution set. The PSEQ detects the type of instructions (such as DALU or AGU), and the AGU instructions are decoded.

5.1.1.3 Address Generation

The address generation pipeline stage is implemented in the AGU and DALU. In the DALU, the address generation stage includes decoding the DALU instructions. However, in the AGU, the address generation stage includes updating the address pointers as well as the actual memory accesses (driving the address and the read/write strobes). The AGU is also responsible for address calculation when a change-of-flow operation takes place.

5.1.1.4 Execution

During the execution stage, all DALU arithmetic calculations are performed by:

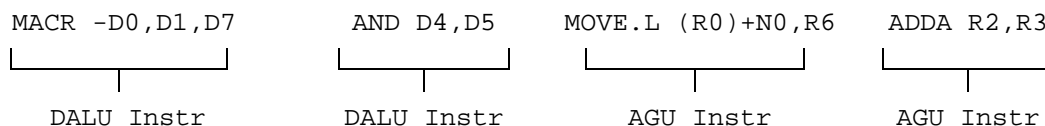
- Reading the data operands from source registers
- Performing arithmetic operations on the data
- Writing the results to destination registers

5.2 Instruction Grouping

The SC140 instruction set architecture is built around a 16-bit instruction set for optimal code density and performance. The core contains two AAUs and four ALUs that enable two instructions to the AAUs and four instructions to the ALUs per clock cycle. The grouping of these instructions is specified explicitly in the assembly source code and encoded by the assembler, subject to the encoding rules described later in this section.

Example 5-1 shows an execution set containing the following four SC140 instructions: a MAC, an AND, a memory read, and an AAU calculation. All four instructions execute independently in a single cycle.

Example 5-1. Four SC140 Instructions in an Execution Set

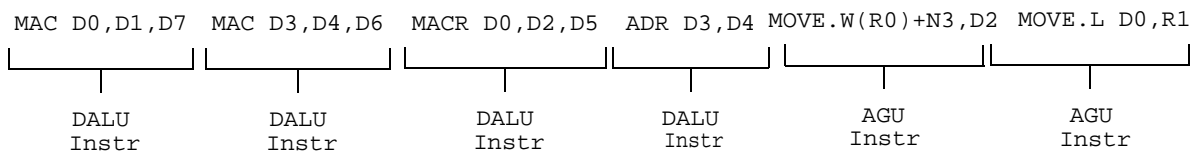


In the execution set above, the four SC140 instructions are grouped. When executed, the following occurs:

1. The contents of the D0 and D1 registers are multiplied fractionally. The result is subtracted from the D7 data register. The final result is then rounded and stored in the D7 data register.
2. The contents of the D4 and D5 registers are ANDed together. The result is stored in the D5 data register.
3. The contents of the 32-bit memory location (pointed to by the R0 register) are moved into the R6 register.
4. The address in the R0 register is incremented by the contents of the N0 register.
5. The contents of R2 are added to the R3 register. This result is stored back in the R3 register.

A second case is illustrated in Example 5-2, which shows a six-instruction execution set that executes in one clock cycle.

Example 5-2. Grouping Six SC140 Instructions in an Execution Set



In the execution set described above, six SC140 instructions are grouped together. When executed, the following occurs:

1. The contents of the D0 and D1 registers are multiplied fractionally. The result is added to the content of the D7 data register. The final result is then stored in the D7 data register.
2. The contents of the D3 and D4 registers are multiplied fractionally. The result is added to the content of the D6 data register. The final result is then stored in the D6 data register.
3. The contents of the D0 and D2 registers are multiplied fractionally. The result is added to the content of the D5 data register. The final result is then rounded and stored in the D5 data register.
4. The contents of the D3 and D4 registers are added together. The result is rounded and stored in the D4 data register.
5. The contents of the 16-bit memory location pointed to by the R0 register is sign extended and moved into the D2 register.
6. The content of the N3 register is added to the content of the R0 register. The result is stored back in the R0 register.
7. The 32 least significant bits of the D0 register are moved to the R1 register.

5.2.1 Grouping Types

The SC140 grouping includes two types of encoding:

- Serial (non-prefix) grouping, which encodes in the two most significant bits (MSB) of instructions.
- Prefix grouping, which encodes a one-word or two-word prefix at the start of the execution set.

The Program Dispatch Unit (PDU) in the PSEQ determines which instructions in each clock cycle should be issued to the execution units. It does this by decoding the grouping information.

In serial grouping, the value 00 in the two most significant bits (MS) of an instruction word indicates that this word is to be grouped with the next instruction word. An instruction with a value other than 00 in its two MS bits is considered the last instruction in the set, and marks the execution set boundary.

In prefix grouping, if a prefix exists at the beginning of an execution set, the PDU uses it to determine the grouping information, including the number of instruction words grouped in the execution set.

Figure 5-2 illustrates the serial and prefix methods for the SC140 grouping mechanism:

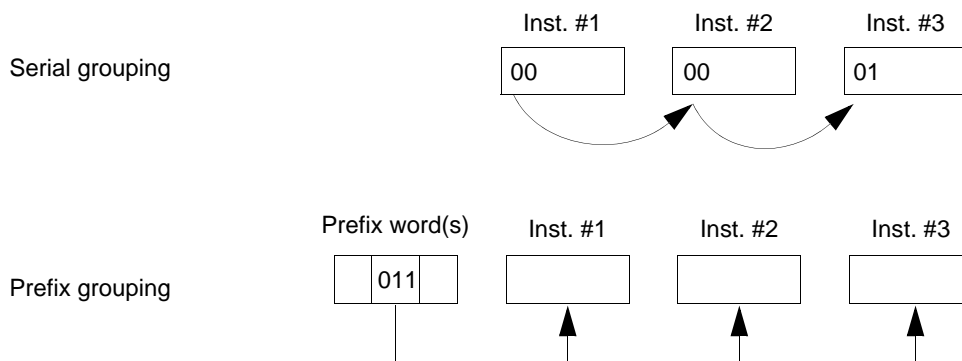


Figure 5-2. Instruction Grouping Methods

Prefix grouping can group together any instructions that have available execution units. However, the prefix method requires one additional instruction word per execution set. Serial grouping is more compact, but only supports a subset of instructions. The assembler automatically selects serial or prefix grouping based on the instructions in each execution set so that the encoding length is minimized. The grouping method selection algorithm is described in Figure 5-3.

5.2.1.1 Serial Grouping

In the serial grouping method, the two most significant bits of each instruction in the execution set provide the core with the necessary information to perform instruction grouping.

Each SC140 instruction belongs to one of the following four types:

- **Type 1** — Basic DALU and move instructions, which are frequently used single-word instructions.
- **Type 2** — Additional DALU, move, and AGU arithmetic instructions that are also single-word instructions, but not used as frequently as Type 1 instructions.
- **Type 3** — Two-word and three-word DALU, move, and AGU arithmetic instructions.
- **Type 4** — All other instructions, which may be one or two words long.

The serial grouping options for an execution set are:

- One to six Type 1 instructions.
- One Type 2 instruction grouped with up to five Type 1 instructions, on condition that a Type 2 instruction can be the last in VLES. Refer to [Section 7.4.1.3, “Assembler Reordering,”](#) on page 7-3.
- One Type 3 instruction grouped with up to five Type 1 instructions, on condition that a Type 3 instruction can be the last in VLES. Refer to [Section 7.4.1.3, “Assembler Reordering,”](#) on page 7-3.
- One Type 4 instruction.

The two MS bit combinations that characterize serially grouped Type 1 instructions are:

- 00 for instructions grouped with the next instruction.
- 01 to indicate the last Type 1 instruction in the set.

In serial grouping (by definition), Type 2, 3, and 4 instructions terminate an execution set. Type 4 instructions cannot be grouped with any other instruction.

5.2.1.2 Prefix Grouping

The SC140 architecture uses prefix words to encode architecture extensions for an entire execution set. The prefix word is a part of the execution set but is not issued directly to any of the execution units.

The grouping information encoded in the prefix includes the number of words to be grouped (including the number of prefix words) minus one. Valid values are from 0 to 7. A value of 0 corresponds to a NOP instruction that is not dispatched.



5.2.2 Prefix Types

The SC140 architecture supports 2 types of prefix instructions, each is used to convey a subset or all of the following information about the VLES:

- The number of instructions that are grouped together in the execution set.
- Conditional execution of the whole set or a subgroup of the set (encoding the IFT/IFF/IFA prefix instructions).
- Looping information for supporting hardware loops (encoding the LPMARKA and LPMARKB bits).
- Encoding extensions for high register banks (D8-D15, R8-R15).

The prefix instructions use either one or two instruction words. Since the fetch set is eight words long, and the maximum issue width is six (four DALU instructions and two AGU instructions), there is usually room for two prefix words without affecting performance. However, in order to save code size, 3 prefix instruction types were defined. Two one-word prefix types have a subset of the mentioned functionality. A two-word prefix has all of the listed functionality. The selection of the right prefix type is done by the assembler which automatically chooses the smallest prefix type (or no prefix at all); see Figure 5-3.

The detailed encoding for prefix words is specified in [Appendix A.1.5, “Prefix Word Encoding.”](#)

5.2.2.1 Two-Word Prefix

The two-word prefix includes all information that could be specified in a prefix:

- Number of instructions in the VLES
- Mark hardware loop information
- Specify conditional execution of the VLES or sub-groups of the VLES
- Encode high register banks (D8-D15, R8-R15)

The SC140 16-bit instruction encoding has a three bit field for specifying each data register or address pointer register. On their own, these instructions can encode eight DALU registers (D0–D7) and eight address pointers (R0–R7). In order to specify operands that belong to the high register banks (D8–D15, and R8–R15), additional register field bits are encoded in a second prefix word.

The two-word prefix includes a register field for each execution unit in the core (namely, four fields for DALU instructions and two fields for AGU instructions). At most, DALU instructions have three operands (for example, ADD D0,D1,D2). Therefore, each DALU field is three bits, so that each operand can be independently specified to be in the high bank. Most AGU instructions have two operands (for example, MOVE (R0)+,D0). Therefore, each AGU field has two bits.

A register extension bit is added for each possible operand in each execution unit. If this bit is set, it signifies that the respective operand uses a register from the high bank. If this bit is cleared, or if the respective set does not include a two-word prefix, the operand uses a register from the low bank. A two-word prefix is generated by the assembler if at least one of the instructions in the execution set uses a register from the high bank.

For a description of what conditional execution options are available, see [Section 5.2.3, “Conditional Execution.”](#)

For a description about the function of HW loop support with LPMARK, see [Section 5.4, “Hardware Loops.”](#)

5.2.2.2 One-Word Low Register Prefix

The One-Word Low register prefix encodes all information of the two-word prefix, except for encoding for high registers. It is used whenever no instruction in the VLES uses a high register, and a prefix is needed for one of the other reasons: instruction grouping that cannot be done with serial grouping, conditional execution of the whole VLES or a sub-group, and encoding for HW loop support.

5.2.3 Conditional Execution

Certain instructions are executed conditional on the state of the T-bit in the status register (SR). For example, JT (Jump if True) is executed only if the T-bit is set. The SC140 also supports conditional execution of a group or subgroup of instructions in an execution set. A group or subgroup represents the instructions that may be conditionally executed depending upon a single condition encoded in the prefix. The single condition is specified by the following prefix instructions:

Table 5-3. Prefix Instructions

Instruction	Description
IFT (if true)	Execute the group or subgroup if the T bit is set
IFF (if false)	Execute the group or subgroup if the T bit is clear
IFA (if always)	Execute the group or subgroup unconditionally

For example:

```
IFT    ADD D0,D1,D2    MOVE.L (R0)+,D0
```

The set as a whole (including the ADD and MOVE instructions) is executed only if the T-bit is set. The instructions in the subgroups may themselves be conditional. For example, using MOVEc, TFRc and Jc can add further conditional control. However, the subgroups themselves may not contain another IFC instruction.

If no IFC instructions exist in the execution set, the default is the unconditional execution of the whole set.

For finer control, it is possible to split the instructions in the execution set into two subgroups 1 and 2, conditionally controlling the execution of each subgroup independently. Refer to [Section 5.2.5, “Instruction Reordering Within an Execution Set,”](#) for information about how the assembler reorders instructions for encoding conditional execution sets.

Table 5-4 displays the conditional IFC syntax. In the table, [inst] represents optional additional instructions. Refer to [Section 7.5.3, “Prefix Grouping Rules,”](#) on page 7-11 for the IFC programming rules.

Table 5-4. Conditional IFC Syntax

Assembly Syntax	Meaning
[IFA] inst [inst]	Unconditional execution of the VLES



Table 5-4. Conditional IFc Syntax

Assembly Syntax	Meaning
IFT inst [inst] IFF inst [inst]	Execution of subgroup1 if T==1 Execution of subgroup2 if T==0
IFT inst [inst]	Execution of the whole group if T==1
IFF inst [inst]	Execution of the whole group if T==0
IFT inst [inst] IFA inst [inst]	Execution of subgroup1 if T==1; always execute subgroup2
IFF inst [inst] IFA inst [inst]	Execution of subgroup1 if T==0; always execute subgroup2

5.2.4 Prefix Selection Algorithm

The grouping method (or encoding of prefix words) is not specified by the programmer. The assembler analyzes each execution set and attempts to group the instructions in a way that minimizes the number of instruction words. If possible, serial grouping is chosen. However, if extra grouping information is necessary, a one-word prefix is generated. A two-word prefix is generated only when high register banks are used in the execution set. The assembler encodes the execution set according to these principles, as shown in Figure 5-3.

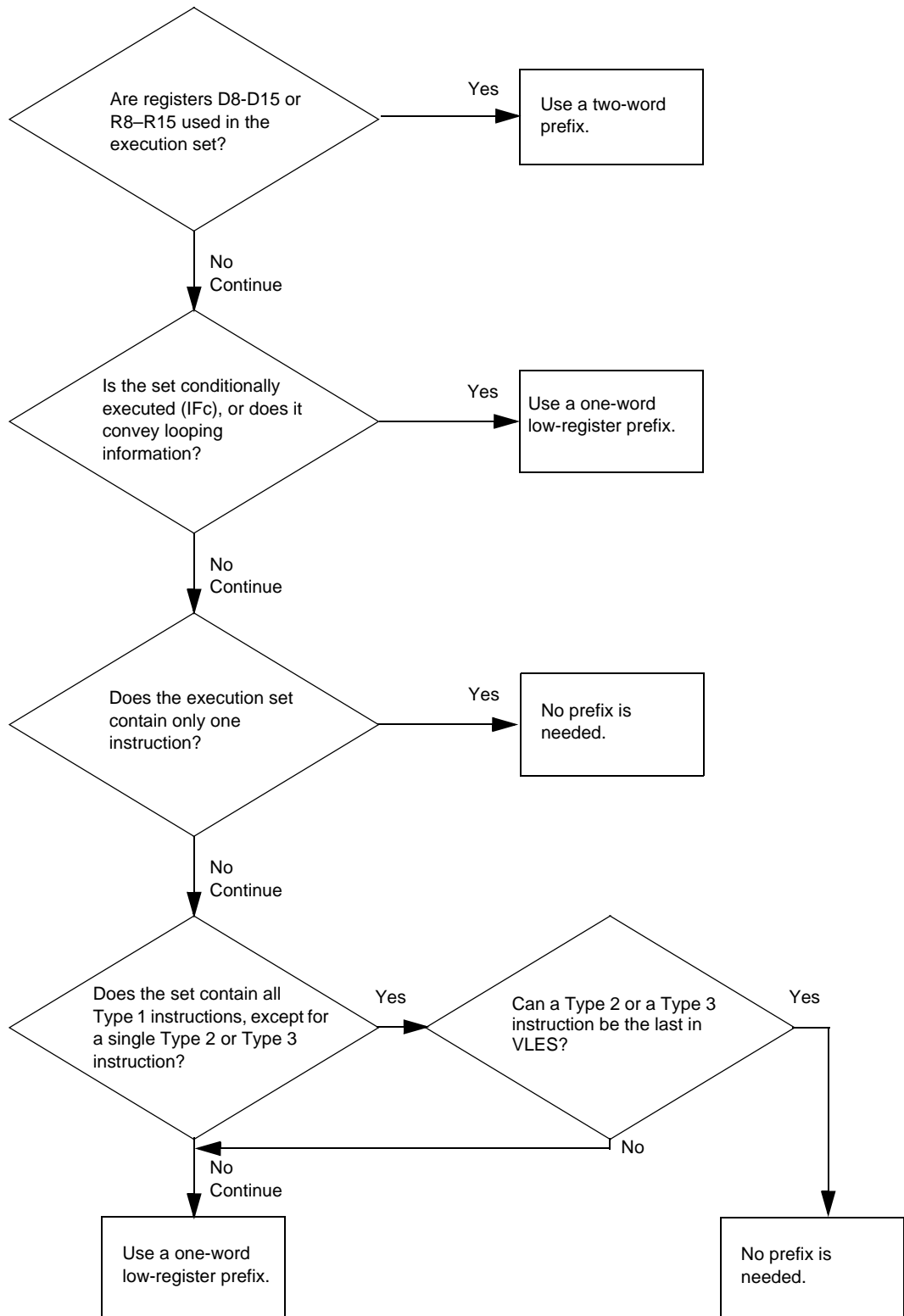


Figure 5-3. Low Register Prefix Selection Algorithm



5.2.5 Instruction Reordering Within an Execution Set

The SC140 can execute up to four DALU instructions and up to two AGU instructions concurrently. These instructions are grouped together in an execution set and dispatched in parallel to the execution units by the PDU. Since the execution units of each type are identical (in principle), any ALU can receive any DALU instruction. As well, any AAU can receive any AGU instruction. The hardware takes advantage of this fact to reduce internal routing from the PDU to the execution units. As a result of this reduction, some reordering may be necessary concerning instruction positions within an execution set.

In general, execution set reordering is transparent to application developers. The assembler appropriately reorders the instruction encoding in an execution set. However, the assembler's behavior may become apparent upon disassembly of the binary code when the order of instructions in the set may be different from the source code. In some rare cases, the assembler may add a NOP instruction in order to accomplish the reordering.

An execution set can include up to eight instruction words, occupying positions 0 through 7 within the set. The position of a multi-word instruction is defined as the position of its first word. Example 5-3 shows the positions occupied by 3 one-word (1w) instructions and 2 two-word (2w) instructions grouped with a one-word prefix:

Example 5-3. Execution Set with Three One-word and Two Two-word Instructions

Position	0	1	2	3	4	5	6	7
	1w prefix	2w - - ext	1w	2w - - ext	1w	1w		

The instruction reordering by the assembler operates as follows:

- Instruction words of an execution set must be encoded contiguously. No encoding gaps are allowed.
- Up to two AGU instructions may appear in an execution set. One must encode in an even position. The other must encode in an odd position. If there is only one AGU instruction, it can be encoded anywhere.
- Up to four DALU instructions may appear in an execution set. These DALU instructions must encode in different positions in modulo 4 arithmetic. For example, two DALU instructions in the same set cannot be encoded in positions 0 and 4, positions 1 and 5, and so on.
- An execution set with a prefix can contain up to 2 two-word instructions. One two-word instruction must encode in an even position and the other in an odd position. Thus, the 2 two-word instructions of an execution set must be encoded with an odd number of instruction words between them.
- Some execution sets contain two conditional subgroups using IFT/IFF/IFA instructions:

Example 5-4. Conditional VLES Having Two Subgroups

```
IFT ADD D0,D1,D2 RND D2,D3 IFF SUB D0,D1,D2 ADDA R0,R1
```

Instructions within a conditional VLES are assigned to two subgroups as follows:

- **Subgroup1** — Instructions encode in the even word positions.
- **Subgroup2** — Instructions encode in the odd word positions.

This means that in any subgroup one cannot have more than one two-word instruction. (since according to the previous bullet one two-word instruction should be in even place and the other in odd).

In this example, instructions of the IFT subgroup encode in subgroup1 (even word positions) while instructions of the IFF subgroup encode in subgroup2 (odd word positions) of the VLES. The assembly syntax completely hides this interleaved encoding from the programmer.

- In cases where more than one DALU instruction in the execution set affects the carry bit C in SR (according to the instruction definition), the last (right or bottom in the assembly source code) carry-updating instruction that actually executes updates the carry bit while the other instructions do not affect the carry bit. If no carry-affecting instructions execute, the carry bit is not affected. The assembler keeps the last carry-affecting instruction as the last (highest position) carry-affecting instruction in the VLES encoding. For two IFc subgroups, this encoding rule applies independently to each subgroup encoding.

In some cases, the assembler adds a NOP instruction during the encoding process. For example, if 2 two-word instructions are grouped, they must be separated by an odd number of instruction words. If no one-word instructions are included in the set, the assembler inserts a NOP instruction. Example 5-5 illustrates such a case.

Example 5-5. Set of 2 Two-word Instructions Requiring a NOP

```
MOVE #xxxx,D0 MOVE #xxxx,D1
```



Given the execution set in Example 5-5, the assembler adds a NOP to the object code for correct encoding.

5.3 Instruction Timing

Most of the instructions used for DSP algorithms take one cycle to execute. They can be grouped together and executed simultaneously. Other instructions, such as those used in the control portion of the application, may take more than one cycle to execute. Some of these multi-cycle instructions are change-of-flow (COF) instructions. Other control-oriented instructions use special addressing modes, or perform read-modify-write operations on memory.

Most sequential (non-change-of-flow) instructions take one cycle to execute. They include DALU, AGU arithmetic, and data moves with simple addressing modes. Data moves with address pre-calculation take two cycles, and atomic read-modify-write BMU instructions take two or three cycles.

Change-of-flow (COF) instructions take three or more cycles to execute. They include direct, PC-relative, conditional, delayed jumps and branches, and loop control instructions.

Parallel execution takes place when two or more instructions (grouped into an execution set) execute simultaneously. Instructions belonging to an execution set always start execution concurrently. A set of instructions start execution only after all the instructions belonging to previous execution sets are completed. Therefore, an execution set's execution time is determined by the instruction in the set that has the longest execution time.

This section describes the time needed to execute SC140 instructions as measured in clock cycles. In the discussion below, it is assumed that memory accesses are zero wait-state and contention-free, unless explicitly stated otherwise. This timing is for the current SC140 implementation, and may change with future implementations.

Interrupt timing and memory access timing is also discussed in this section.

Table 5-5 summarizes the timing of the various categories of SC140 instructions.

Table 5-5. Instruction Categories Timing Summary

Basic Instruction Category	Example/Condition	Number of Clock Cycles
DALU	MAC D0, D1, D2	1
Data move with simple addressing	MOVE.W (R0)+N2, D3	1
Data move with address pre-calculation	MOVE.W (R5+N0), D4	2
BMU with simple addressing	BMSET.W # $\$1010$, (R0)	2
BMU with address pre-calculation	BMSET.W # $\$1010$, (SP+ $\$10$)	3
Direct change-of-flow	JMP dest	3
PC-relative change-of-flow	BRA dest	4
Conditional change-of-flow	If condition is true	4
	If condition is false	1
Delayed change-of-flow	Direct	3 (some cycles used by the execution set in the delay slot)
	PC-relative	4 (some cycles used by the execution set in the delay slot)

5.3.1 Sequential Instruction Timing

This section describes the timing of non-COF instructions:

- All DALU instructions take one clock cycle to execute.
- All AGU arithmetic instructions take one cycle to execute.
- All memory MOVE instructions (zero-wait-states without contention) take one clock cycle to execute, unless the addressing mode needs to perform a pre-calculation, in which case, the move executes in two cycles. For example, the move instructions below take two cycles:
 - MOVE.L d0, (Rn + N0)
 - MOVE.L d0, (Rn + $\$5$)
 - MOVE.L d0, (Rn + Rm)
 - MOVE.L d0, (SP + $\$100$)
- All bit mask (BMU) instructions execute in two cycles on registers and memory (zero-wait-states without contention) with simple addressing modes. However, if a pre-calculation is required, such as an SP offset, a third cycle is added.



5.3.1.1 DALU Instruction Timing

DALU instructions are the most timing-critical instructions in the DSP algorithm kernels, taking only one cycle to execute. DALU instructions consist, among others, of the following:

- Multiply-accumulate (MAC)
- Multiply (MPY)
- ADD
- SUB
- Compare
- Shift
- Test

5.3.1.2 Move Instruction Timing

Most of the move instructions take one cycle to execute, assuming a zero-wait-state, contention-free memory. The exception is for the addressing modes requiring an arithmetic calculation of a new address: $(Rn + N0)$, $(Rn + Rm)$, $(Rn + x)$, $(Rn + xxxx)$, $(SP - xx)$ and $(SP + xxxx)$. These addressing modes require one additional clock cycle to calculate the address of the memory access. All the other versions of data moves are one cycle, including the versions for byte, word, two-word, long-word, four-word, and two long-word operands (signed or unsigned). Data can be moved between memory and register, or between registers.

5.3.1.3 Bit Mask Instruction Timing

The SC140 core includes various instructions for bit mask operations. These instructions are helpful when several bits need to be changed or tested at the same time. The bit mask instructions include the following:

- Bit mask set (BMSET)
- Bit mask clear (BMCLR)
- Bit mask change (BMCHG)
- Bit mask test (BMTSTS, BMTSTC)
- Bit mask test and set (BMTSET)

Bit mask instructions are a read-modify-write instruction. This means they have three steps:

1. Read the operand.
2. Change (set, clear, or change) selected bits.
3. Write the operand back to the original location.

This type of instruction takes two clock cycles to execute for the simple addressing modes, and three clock cycles for the addressing modes that require pre-calculation of the address.

Refer to [Appendix A, “SC140 DSP Core Instruction Set,”](#) for a full description of the bit mask instructions.

5.3.2 Change-Of-Flow Instruction Timing

The change-of-flow (COF) instructions include branches, jumps, traps, returns, conditional branches, conditional jumps, and loop control instructions that affect the program counter and/or software stack. Program control instructions may affect or be affected by status register bits as specified in the instruction. In the SC140 instruction set naming convention, “jump” signifies instructions using a direct destination address (either absolute or in a register), while “branch” signifies instructions that use a PC-relative offset to specify the destination address.

Jumps and branches to subroutines (JSR/BSR) include implicit push operations to the stack. Similarly, returns from subroutines or exceptions (RTS/RTSTK/RTE) include implicit pop operations from the stack.

COF instructions usually take longer to execute because the pipeline is disrupted during their execution. They are usually most affected by the access time to memory as well as the number of stages in the pipeline. In order to use time more efficiently, most of the COF instructions have a delayed version that enables the execution of one execution set while the pipeline is filling up. The delayed instruction effectively saves one or more cycles over the non-delayed version. The suffix D indicates the delayed version of an instruction, as in JMP and JMPD. For example, JMPD is the delayed version of the JMP instruction.

In Example 5-6, the MOVE.W instruction is logically executed before the delayed jump. The delayed COF instruction JMPD as well as the execution set in the delay slot are a non-interruptible sequence.

Example 5-6. Delayed Change-of-Flow and Its Delay Slot

JMPD	destination_label	;delayed COF
MOVE.W	(R0+N0),D0	;delay slot

COF instructions are of two types - non-loop COF shown in Table 5-6, and loop COF shown in Table 5-7.

Table 5-6. Non-Loop Change-of-Flow Instructions

Instruction	Description
BF	Branch if false
BFD	Branch if false (delayed)
BRA	Branch
BRAD	Branch (delayed)
BSR	Branch to subroutine
BSRD	Branch to subroutine (delayed)
BT	Branch if true
BTD	Branch if true (delayed)
JF	Jump if false
JFD	Jump if false (delayed)
JMP	Jump



Table 5-6. Non-Loop Change-of-Flow Instructions (Continued)

Instruction	Description
JMPD	Jump (delayed)
JSR	Jump to subroutine
JSRD	Jump to subroutine (delayed)
JT	Jump if true
JTD	Jump if true (delayed)
RTE	Return from exception
RTED	Return from exception (delayed)
RTS	Return from subroutine
RTSD	Return from subroutine (delayed)
RTSTK	Restore PC from the stack, updating SP
RTSTKD	Restore PC from the stack, updating SP (delayed)
TRAP	Execute a precise software exception

Table 5-7. Loop Change-Of-Flow Instructions

Instruction	Description
BREAK	Terminate the loop and branch to an address
CONT	Jump to the start of the loop to start the next iteration
CONTD	Jump to the start of the loop to start the next iteration (delayed)
SKIPLS	Test the active LC and skip the loop if it is equal or smaller than zero

5.3.2.1 Direct, PC-Relative, and Conditional COF

The SC140 core implements a five-stage pipeline with two stages dedicated to memory access. This results in the addition of two clock cycles for unconditional COF instructions that use immediate values as well as the addition of three clock cycles for the PC-relative COF instructions. Conditional change-of-flow instructions, where the condition is true (meaning the change-of-flow operation is taken), always take an additional three cycles. When a conditional change-of-flow is determined as not taken (meaning the condition is false), there are no additional cycles.

5.3.2.2 Delayed COF

When a change-of-flow instruction is executed, the core must wait for the pipeline to fill, starting with a new pre-fetch from memory. A delay slot is the next VLES after a delayed change-of-flow instruction. Since it is possible to use the delay slots of the change-of-flow operation to continue the execution of the previously fetched instructions, special delayed instructions are added to the instruction set. These instructions use part or all of the delay cycles to execute one additional execution set. This effectively reduces the penalty for utilizing a change-of-flow operation. If the additional execution set in the delay slot is included in the cycle count, the number of cycles for the change-of-flow instruction are effectively reduced. Refer to [Section 5.3.2, “Change-Of-Flow Instruction Timing,”](#) on page 5-17 for further details.

5.3.2.3 COF Execution Cycles

The basic change-of-flow JMP instruction takes three cycles to execute. However, the number of cycles is different for the following change-of-flow instructions:

- PC-relative instructions such as BRA require an additional cycle to calculate the destination.
- Delayed instructions such as JMPD effectively require the same cycle count as the non-delayed version (in this example JMP) minus the execution cycle count of the set in the delay slot. This is the case because the pipeline fill-up time is used to execute a useful execution set. The actual time taken to jump to the new address is the same for the delayed or non-delayed version. However, the effective cycle count is less for the delayed version since the execution of the instructions in the delay slot would be extra counts if the non-delayed version was used.

The delay slot lasts for the full execution time of the set in the delay slot, which may be more than one cycle. The minimum execution time of a delayed instruction is one cycle. For example:

```
JMPD dest;    takes 1 cycle (3-2=1), because the next instruction
MOVE.W d0,(sp + xxx); takes 2 cycles
```

Stalls that originate in delay slot instructions, and are caused by a memory access wait-state or contention, stall the whole core, and are not deducted from the cycle count.

- Conditional change-of-flow instructions (JT/JF/BT/BF) require four cycles to execute (if taken), and one cycle to execute (if not taken).
- The core implements a mechanism for fast return from subroutine. The return address of subroutines is kept in a hidden return address stack (RAS) register in addition to being pushed to the stack. This saves the need to read it from the stack in memory upon return. However, this hidden register is not valid if there was another jump to a subroutine before the return, in which case, the core adds two cycles to the RTS instruction to read the return address from the stack. Refer to [Section 5.5.5, “Fast Return from Subroutines,”](#) for a more detailed description of the fast return mechanism.
- The core keeps a “shadow” version of SP-8 to save pre-calculation time in case of a POP. If SP was explicitly changed by a TFRA or an AGU arithmetic instruction, the shadow SP is not valid and another cycle is needed for the first POP pre-calculation (or equivalent, such as RTE). Refer to [Section 5.5.4, “Shadow Stack Pointer Registers,”](#) for a more detailed description of the shadow SP mechanism.
- A change-of-flow instruction (jump, branch, interrupt, or long loop iteration) made to an execution set destination that is spread over two fetch sets, requires an additional cycle for memory access. An execution set is not necessarily aligned to a fetch set, and can overlap two fetch sets. The core keeps two fetch sets in a buffer, so this is not normally a problem. However, when a



change-of-flow occurs to a new execution set spread over two fetch sets, two new fetches must be read from memory.

- The subroutine call instructions (JSR, JSRD, BSR, and BSRD) need one free cycle in order to push the return PC and SR onto the stack. Normally, a subroutine call instruction uses one of the idle cycles while the pipeline is filling up so that no stall occurs. However, one stall cycle is added if the instructions that execute in parallel with the subroutine CALL need more cycles than a specific number. In essence, an additional cycle is added to a subroutine call instruction when $(C_{jn} + C_d) \geq C_j$ where:
 - C_{jn} Highest cycle count of instructions grouped with CALL
 - C_j Cycle count of the non-delayed version of CALL (for example, BSR and BSRD, $C_j = 4$)
 - C_d Cycle count of the set in the delayed slot (if CALL is not a delayed instruction, $C_d = 0$)

Example 5-7 shows a case when a stall cycle is added.

Example 5-7. Subroutine Call Timing

```

JSRD _subr          MOVE.W (R0+2),D0    ; Cj = 3, Cjn = 2
ADDA R0,R1          ; Cd = 1
    
```

Table 5-8 summarizes the cycle count for change-of-flow instructions. In the Number of Cycles column, C_d represents the length of the delay slot in cycles. The technique of subtracting the cycles of the delay slot instructions from the cycle count of the delayed change-of-flow instruction assumes that the delay slot instructions' cycles are counted separately. The net count should be zero since the instructions are "hidden" in the delay slot. The minimum number of cycles is specified for the delayed instructions, but only when the number of cycles is small enough for the minimum number of cycles to actually occur. If no number appears in the Minimum Number of Cycles column, the equation in the Number of Cycles column applies, with no minimum.

Table 5-8. Number of Cycles Needed by Change-of-Flow Instructions

Instruction	Number of Cycles	Minimum Number of Cycles	Condition
JMP	3		
JMPD	$3 - C_d$	1	
JSR	3 4		$C_{jn} < 3$ $C_{jn} \geq 3$
JSRD	$1 + C_{jn}$		
BRA, BSR	4		
BRAD	$4 - C_d$	1	
BSRD	$4 - C_d$ $1 + C_{jn}$	2	$C_{jn} + C_d < 4$ $C_{jn} + C_d \geq 4$
Jc/Bc	4 1		Jump is taken. Jump is not taken.
JcD/BcD	$4 - C_d$ 1		Jump is taken. Jump is not taken.

Table 5-8. Number of Cycles Needed by Change-of-Flow Instructions (Continued)

Instruction	Number of Cycles	Minimum Number of Cycles	Condition
RTE	5 6		Shadow SP is valid. Shadow SP is not valid.
RTED	$5 - C_d$ $6 - C_d$		Shadow SP is valid. Shadow SP is not valid.
RTS	3 5 6		RAS is valid. RAS is not valid and shadow SP is valid. RAS is not valid and shadow SP is not valid.
RTSD	$3 - C_d$ $3 - C_d$ $5 - C_d$ $6 - C_d$	1 2	RAS is valid and shadow SP is valid. RAS is valid and shadow SP is not valid. RAS is not valid and shadow SP is valid. RAS is not valid and shadow SP is not valid.
RTSTK	5 6		Shadow SP is valid. Shadow SP is not valid.
RTSTKD	$5 - C_d$ $6 - C_d$		Shadow SP is valid. Shadow SP is not valid.
SKIPLS	4 1		Jump is taken. Jump is not taken.
BREAK	4		
CONT	3 4		SA is taken. Destination is taken.
CONTD	$3 - C_d$ $4 - C_d$	1 1	SA is taken. Destination is taken.
TRAP	5		

5.3.3 Memory Access Timing

The SC140 core executes up to one execution set per cycle. The programmer can specify up to two memory MOVE instructions per execution set. Since the memory interface has one program and two data buses, up to three simultaneous memory accesses can occur as described in [Section 2.4, “Memory Interface.”](#)

The memory organization determines when memory contention occurs for simultaneous accesses. There is likely contention if the same byte-addressed location is accessed. However, there could be a contention in other cases, due to the memory internal structure. Because memory is not implemented to provide true multi-port access, accessing of two different addresses in the same memory block may cause a contention. The intent of the following section is to describe the timing for memory accesses generated by instructions in the same execution set. In some examples, no problems arise since the memory accesses fall into different cycles. In other examples, memory contention can occur.



The read or write for each memory access can be mapped to the execution cycle in which they operate as follows:

- Cycle 1
 - Move read or write without address pre-calculation.
 - Bit mask read without address pre-calculation.
 - Pop read with shadow SP valid.
- Cycle 2
 - Move read or write with address pre-calculation.
 - Bit mask read with address pre-calculation.
 - Bit mask write without address pre-calculation.
 - Pop read with shadow SP invalid.
- Cycle 3
 - Bit mask write with address pre-calculation.

Contention may occur when two instructions in an execution set attempt to access the same physical memory module in the same cycle. The memory system evaluates contention on a cycle-by-cycle basis, not for the execution set as a whole.

The following conventions apply to the execution of memory access operations:

- Each AGU access operation is performed in its cycle number, independent of any other access.
- Operations that execute in different cycles are performed in cycle sequence without contention.
- Bit mask instructions (such as BMSET, BMCLR, and BMCHG) are read-modify-write instructions. These instructions each generate two memory accesses in sequence.
- Pop instruction timing depends on the validity of the stack pointer (SP) shadow register that holds a pre-decremented value of the SP in order to avoid the need for pre-calculation. If shadow SP is not valid (for example, after an explicit SP update), another cycle is needed for the first pop in order to perform the pre-calculation.

The following rules apply to cases of contention due to dual access to the same physical memory module by two instructions in the same cycle:

- A memory read instruction executes before a memory write instruction.
- For two memory writes to different locations, the order is undefined, meaning that it is implementation-specific. The program algorithm should not assume any specific behavior of the memory system for the memory writes.
- Two memory writes to the same location in the same execution set is not allowed by the SC140 programming rules. If this occurs, the memory results are undefined.

5.3.3.1 Memory Access Examples

This section describes the contention cases of two memory access instructions grouped in a VLES. If two memory writes are grouped in a VLES, it is assumed that the two write addresses contend for the same physical memory module but do not access the same memory locations. Two memory writes to the same location are not allowed per [Section 7.6, “Dynamic Programming Rules.”](#) The following description is assumes a simple memory without write buffers, where the memory system executes all accesses on a

cycle-by-cycle basis. Accesses issued on the same cycle may cause a contention. The cases where contentions will occur and how many stall cycles will be introduced depends on the definition of the memory system, which may be different than that described below.

Example 5-8 provides an execution set that does not cause contention since the instructions execute in different cycles.

Example 5-8. Parallel Execution of Two Move Instructions

```
MOVE.L D0,(R0)           MOVE.B (R1+1),D1;
;Cycle 1: write to memory
;Cycle 2: read from memory, cycle 2 required by the pre-calculation of (R1+1)
```

Example 5-9 provides two cases of parallel execution by a bit mask and write instruction. In the example, it is assumed that the memory accesses are made to addresses that cause contention. In Case A, the read and write operations scheduled for Cycle 1 will cause contention. In Case B, the two write operations in Cycle 2 will cause contention.

Example 5-9. Execution Set Containing a Bit Mask and a Move Instruction

A)

```
BMSET.W #$0008,(R1)     MOVE.W D0,($8200);
;Cycle 1: read from (R1); ;write to ($8200)
;Cycle 2: write to (R1)
```

B)

```
BMSET.W #$0010,(R1)     MOVE.W D1,(R0+2)
;Cycle 1: read from (R1);
;Cycle 2: write to (R1) ;write to (R0+2)
```

Example 5-10 shows the parallel execution of a bit mask instruction and a move instruction that does not cause contention. The write operation from the MOVE instruction occurs at Cycle 1. The BMU read operation occurs at Cycle 2. The BMU write operation that accesses the same location in memory takes place at Cycle 3.

Example 5-10. Execution Set Containing One Bit Mask Instruction

```
BMSET.W #$0080,(SP-2)   MOVE.W D2,($8200)
;Cycle 1: write to ($8200)
;Cycle 2: read from (SP-2)
;Cycle 3: write to (SP-2)
```



Example 5-11 shows the parallel execution of a bit mask and a pop instruction. The example distinguishes the cases of a valid and invalid shadow SP (see [Section 5.5.4, “Shadow Stack Pointer Registers.”](#))

If the shadow SP is not valid as in Case A (meaning the address of the stack pointer was overwritten), the address of the stack pointer (SP-8) must be pre-calculated from the value in SP. There is no contention since the two read operations occur at different cycles.

If the shadow SP is valid as in Case B, the address of the stack pointer that was saved in the shadow SP is readily available. Cycle 1 now includes two reads that may access the same location and may cause contention.

Example 5-11. Execution Set Containing a Bit Mask and a Pop Instruction

A)

```
BMTSTS.W #0800,(R0)          POP D0
;Shadow SP is not valid, R0==SP-8
;Cycle 1: read from (R0)
;Cycle 2: read from (SP-8)
```

B)

```
BMTSTS.W #0800,(R0)          POP D0
;Shadow SP is valid
;Cycle 1: read from (R0); read from (SP-8)
```

5.3.3.2 Implicit Push/Pop Memory Timing

Instructions with implicit push/pop memory access (such as JSR and RTE) execute the memory access after all other accesses in the execution set have been performed.

Delayed instructions with implicit push memory access (such as JSRD) access memory after all other accesses in the delay slot have been performed. Delayed instructions with implicit pop memory access (such as RTSD and so on) access memory before accesses in the delay slot are performed.

Consequently, these instructions do not cause contention when they are executed in parallel with other instructions that access memory.

5.3.3.3 Memory Stall Conditions

The SC140 can generate up to three memory accesses per cycle consisting of one program fetch and two data accesses. The extent to which the specific memory configuration can support various kinds of simultaneous accesses to memory modules may vary from chip to chip. The memory system identifies access combinations (usually by means of a bus controller) that cannot be supported simultaneously. The memory system stalls the SC140, which results in the serialization of the contending accesses. For example, a stall occurs when a memory unit that can support only one access at a time receives a simultaneous request for two data accesses (or for one program access and one data access). Stalls can also occur if the memory itself is not zero-wait-states, which may be a characteristic of the memory technology (such as flash or DRAM), or may occur with off-chip memory.

5.4 Hardware Loops

One of the most important features of a DSP algorithm is efficient loop execution. The SC140 core has a fully optimized looping mechanism, which enables loop execution with up to four levels of loop nesting. The loop programming model is part of the PSEQ programming model, and includes four pairs of registers that specify the start address of the loop as well as the number of times the loop is to be executed.

5.4.1 Loop Programming Model

There are four pairs of loop registers with two registers in each pair:

- Loop start address registers (SA0, SA1, SA2, SA3)
- Loop counter registers (LC0, LC1, LC2, LC3)

Each pair is responsible for a single hardware loop. The functionality of each register pair is described in the sections that follow.

Figure 5-4 shows the hardware loop programming model. This programming model holds the full loop state and can be saved and restored for exception service routines, context switches, or spill/fill operations to support additional nesting levels.

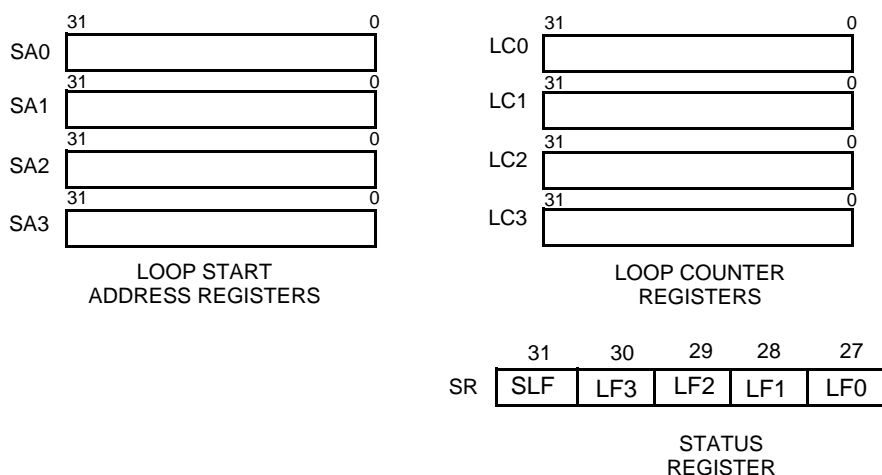


Figure 5-4. Hardware Loop Programming Model

5.4.1.1 Loop Start Address Registers (SAn)

The SAn registers are 32-bit read/write registers that are used to define the address of the first execution set in each loop. The DOSETUPn label instruction initializes the SAn register with the start address. The LOOPSTARTn assembly directive also marks the start address and must be placed at the same address as the label of the DOSETUPn instruction.



5.4.1.2 Loop Counter Registers (LCn)

The LCn registers are 32-bit read/write registers used to define the number of times each loop is to be executed. LCn always holds a 32-bit signed value. This means that the largest number of loop iterations is $2^{31}-1$. The DOENn or DOENSHn instructions initialize the LCn register.

5.4.1.3 Status Register (SR) Loop Flag Bits

Certain status bits in the SR are associated with hardware loop initiation and execution. These bits are set and cleared by special loop instructions such as DOENn as well as various loop conditions. Although not recommended, they can also be set and cleared by explicitly writing the SR register. It is not recommended to explicitly change these bits while a loop is active. The bits are:

- **Loop Flag Bits** — Four loop flag bits (LF0, LF1, LF2, LF3) are defined in the SR, one for each hardware loop. The bit is set when the loop is initiated by either the DOENn or the DOENSHn instruction. It is cleared when the loop terminates.
- **Short Loop Flag Bit** — This bit (SLF) is set when the loop is initiated by the DOENSHn instruction. It is cleared when the loop terminates. The short loop can only be used in the inner-most nesting level.

5.4.2 Loop Notation and Encoding

The notation used in the loop definitions is as follows:

- **Loop Body** — The execution sets that are iterated during loop execution.
- **Long Loop** — A loop body that consists of three or more execution sets.
- **Short Loop** — A loop body that consists of one or two execution sets.
- **Start address (SA)** — The address of the first execution set in a loop body. Do not confuse this with SA0, SA1, SA2, and SA3, which are register names used to hold SA values. The start address is defined by the DOSETUPn label instruction and the LOOPSTARTn assembly directive. These two redundant definitions of the start address must be consistent.
- **Last address (LA)** — The address of the last execution set in a loop body. LA is defined by the LOOPENDn assembly directive. In the case of a loop with only one execution set, SA is also the last address.
- **SA+1** — Address of the execution set following SA (similarly SA+2, and so on).
- **LA-2** — Address of the execution set that comes two execution sets before the execution set at LA (similarly LA-1, and so on).
- **LPMARKA and LPMARKB** — Two marker bits in the prefix words that identify different looping conditions. The LPMARK bits are set automatically by the assembler based on the LOOPSTARTn and LOOPENDn assembly directives, and are not written by the programmer.

Table 5-9 illustrates the location of these marker bits and their functionality in both short and long loops. Refer to [Appendix A, “SC140 DSP Core Instruction Set,”](#) for further details.

Table 5-9. LPMARKA and LPMARKB Bits in Short and Long Loops

Loop Type	LPMARKA		LPMARKB	
	Location	Functionality	Location	Functionality
Short loop	SA	Identifies a single-execution set loop. Causes no timing overhead.	SA	Identifies a two-execution set loop. Causes no timing overhead.
Long loop	LA	Identifies a jump to SA after executing the set at LA, if the loop is repeated. Causes a timing overhead.	LA-2	Identifies a jump to SA after executing the sets at LA-2, LA-1, and LA, if the loop is repeated. Causes no timing overhead.

5.4.3 Loop Initiation and Execution

The following steps are required to initiate a hardware loop:

1. Execute a DOSETUPn instruction at some stage before the loop starts (except in the case of a short loop). This instruction writes the start address of the loop to the corresponding SAN register.
2. Execute a DOENn or DOENSHn instruction to load the corresponding LCn register with the number of iterations for the loop. The corresponding loop flag bit is implicitly set when LCn is loaded with the loop iteration value. The SLF is set if the loop is initialized by a DOENSHn instruction.
3. Execute a SKIPLS instruction before entering the loop to check the value of LCn. If the value of LCn is less than or equal to zero, then the loop is skipped and the program counter (PC) is loaded with the address specified in the SKIPLS instruction. If it is guaranteed that LCn is greater than zero (for example, if the loop is initialized by an immediate value), the SKIPLS instruction can be omitted. The SKIPLS instruction provides the additional flexibility of skipping the steps in the loop completely if the loop count is zero initially.

After the LCn is loaded and the LFn bit is set with the DOENn or DOENSHn instruction, the hardware loop is ready for operation. In long loops, whenever the program reaches the execution set marked by LPMARKB [which appears two execution sets before the last execution set of the loop (LA-2)], LCn is compared to the value one in order to detect loop termination. If the value of LCn is greater than one, the program effectively jumps to the start address while executing the two execution sets at LA-1 and LA. The LCn is decremented by one and the loop is repeated. If the value of LCn is equal or less than one, the loop terminates and the loop flag bit is cleared. Execution of instructions continues in sequence.

In short loops, one or two execution sets are stored in internal buffers and repeated the appropriate number of times according to the value stored in LCn. No program fetches are required for short loops.

5.4.4 Loop Nesting

The core has four hardware loops (LOOP0, LOOP1, LOOP2 and LOOP3) to execute up to four levels of loop nesting. A loop can only be nested within a loop that has a lower index. In a nested loop structure, more than one loop can be enabled at one time. A loop is enabled when its corresponding LFn is set. The LF3–LF0 bits indicate which of the loops are enabled. The enabled loop with the highest index is defined as the “active loop”. Only one loop can be active at a time.

Figure 5-5 shows an example of the loop nesting structure.

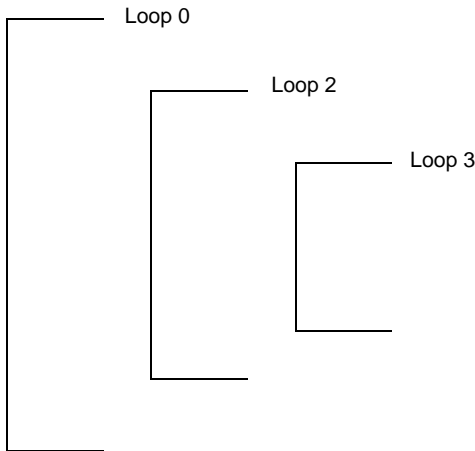


Figure 5-5. Loop Nesting

In Figure 5-5, all three loops are initially disabled. Loop 3 has the highest index, Loop 2 has the next highest index, and Loop 0 has the lowest index of the three. In the normal program flow through the loops, Loop 0 is enabled and its first iteration takes it to Loop 2, which is enabled. Loop 2 has a higher index than Loop 0, so Loop 2 is the active loop. In the first iteration of Loop 2, Loop 3 is enabled and now becomes the active loop. Loop 3 is active until it has finished repeating, at which time Loop 2 becomes active. When Loop 2 stops repeating (including further complete cycles of Loop 3), Loop 0 becomes the active loop. When Loop 0 stops repeating, no loops are active.

5.4.5 Loop Iteration and Termination

The CONT instruction causes the active loop iteration to conditionally terminate before reaching the last execution set of the loop. If the value of LCn is greater than one, then the CONT instruction causes the program to jump to the address stored in SAN. The LCn is decremented by one and the loop is repeated. If the value of LCn is less than or equal to one, then the CONT instruction causes the program to branch to the address specified by the CONT instruction. The LCn is cleared and the loop terminates (LFx is cleared).

The BREAK instruction also causes the active loop to terminate. The program address bus is loaded with the address specified by the BREAK instruction. The loop terminates (LFx is cleared) regardless of the value of LCn, which is not changed.

5.4.6 Loop Control Instructions

Table 5-10 lists the loop instructions.

Table 5-10. Loop Control Instructions

Instruction	Operation
DOSETUPn <label>	Initialize register SAn with <label> address. Used only in long loops.
DOENn (reg or imm)	Activate loop n as a long loop. Performs the following operations: <ul style="list-style-type: none"> • Initializes LCn • Sets LFn in the SR
DOENSHn (reg or imm)	Activate loop n as a short loop. Performs the following operations: <ul style="list-style-type: none"> • Initializes LCn • Sets LFn and SLF in the SR
SKIPLS <label>	If LCn <= 0, jumps to <label>, clearing LFn.
CONT <label>	Within an active loop, if LC > 1, jumps to SAn and decrements LCn. If LC <= 1 jumps to <label> and clears the LCn register as well as LFn.
CONTD <label>	Provides a delayed version of the CONT instruction.
BREAK <label>	Within an active loop, jumps to <label> and clears LFn.

The instructions that activate the loop are either DOENn or DOENSHn. In nested loops, DOENn or DOENSHn must be re-executed in order to re-activate the inner loops. In other words, the DOENn or DOENSHn instruction for an inner loop must be contained in its corresponding outer loop. DOSETUPn is only used to initialize SAn in long loops and need not be re-executed if the value of the SAn register is unchanged. In short loop initialization, DOSETUPn is not needed at all. The instructions SKIPLS, CONT, and BREAK (along with their variants) are optional, to be used only if needed.

The above instructions are assembled from source mnemonics in the conventional way. They are also disassembled normally. In addition, looping bits are encoded in the instruction prefix. When coding a hardware loop in assembly, two loop-related assembly directives must be used to set the LPMARKA and LPMARKB bits as follows:

- **LOOPSTARTn** — Placed immediately before SA.
- **LOOPENDn** — Placed immediately after LA.

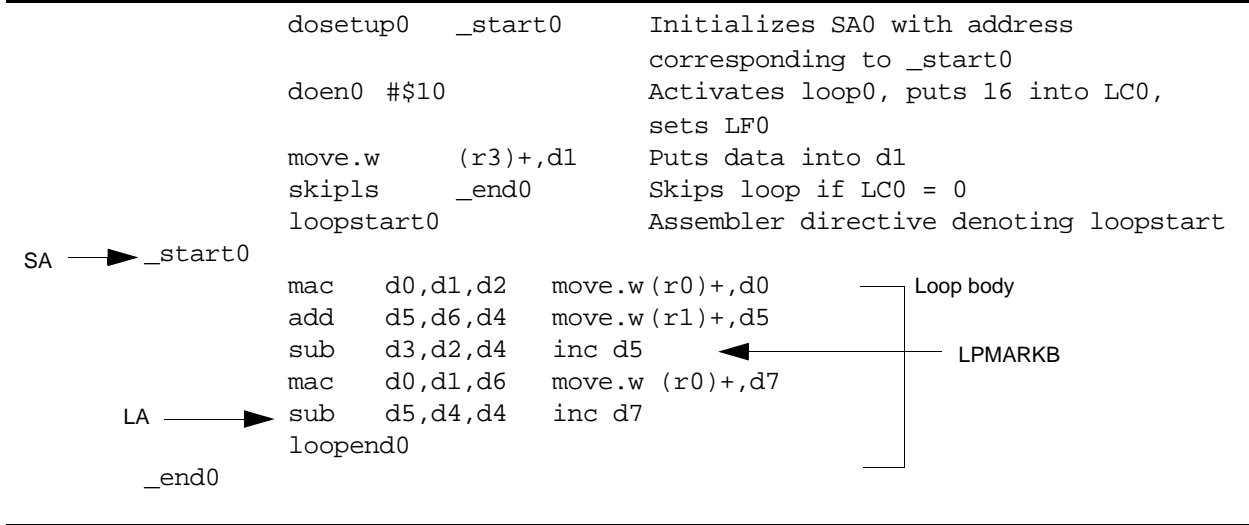
By definition, a loop body n is enveloped by the LOOPSTARTn and LOOPENDn directives.

In disassembled code, the LOOPSTART and LOOPEND directives are not available. The start address information is encoded as an offset in the DOSETUPn instruction for long loops, or in the LPMARK prefix bits for short loops. The last address information is encoded in the LPMARK prefix bits for long loops. The assembler normally places the LPMARKB bit at LA-2. For special cases, such as a SKIPLS

instruction to the last address, the LPMARKA bit will be placed at LA in addition to the LPMARKB bit at LA-2. In nested loops, if the LPMARKA and LPMARKB bits occur in the same execution set, the LPMARKA bit belongs to the inner loop, and the LPMARKB bit belongs to the outer loop.

The following is an example of a long loop.

Example 5-12. Long Loop



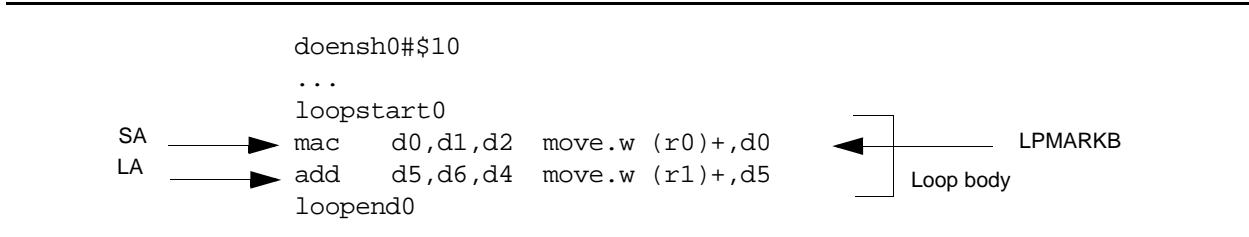
Example 5-13 shows the disassembly of the long loop in Example 5-12.

Example 5-13. Long Loop Disassembly

p:\$00380000	2803	800c	= dosetup0 *+\$c
p:\$00380004	9050		= doen0 #<\$10
p:\$00380006	511b		= move.w (r3)+,d1
p:\$00380008	2103	801a	= skipls >*\$1a
p:\$0038000c	2111	5018	= mac d0,d1,d2 move.w (r0)+,d0
p:\$00380010	2e5a	5519	= add d5,d6,d4 move.w (r1)+,d5
p:\$00380014	94d0	6e3d 66ef	= lpmarkb sub d3,d2,d4 inc d5
p:\$0038001a	2311	5718	= mac d0,d1,d6 move.w (r0)+,d7
p:\$0038001e	2e35	67ef	= sub d5,d4,d4 inc d7

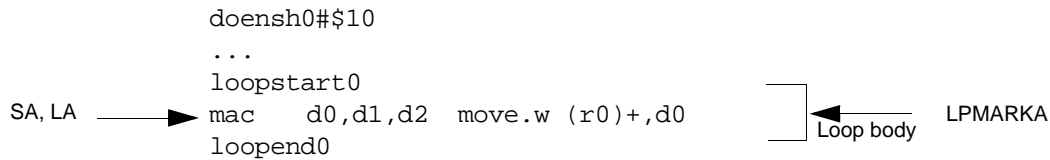
The following is an example of a short loop in two execution sets.

Example 5-14. Short Loop, Two Execution Sets



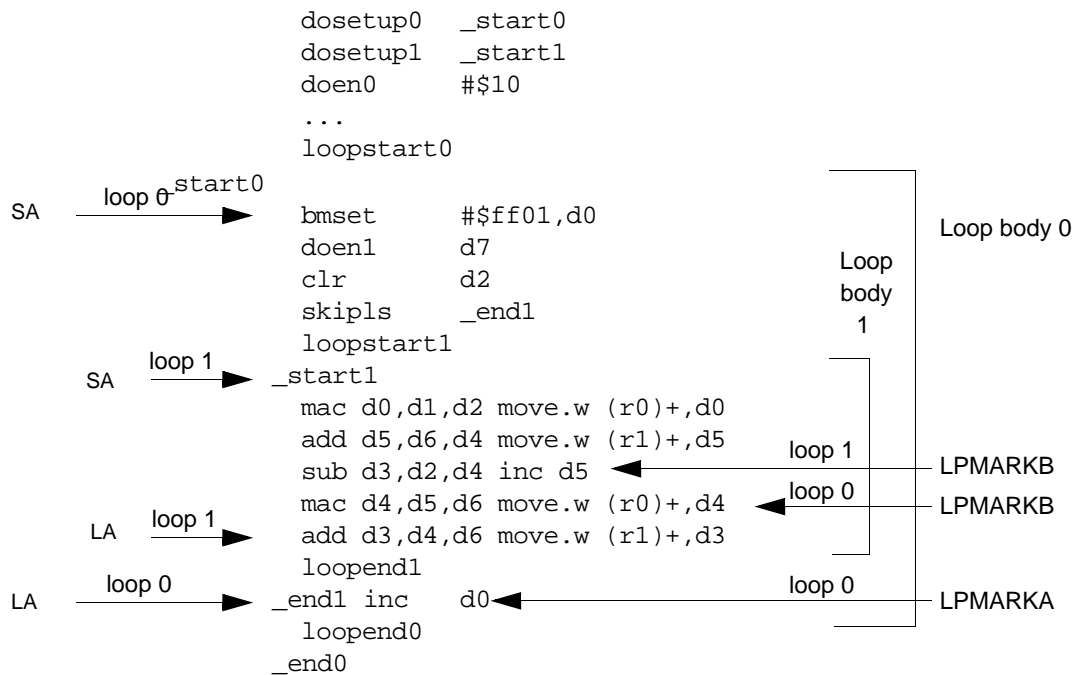
The following is an example of a short loop in one execution set.

Example 5-15. Short Loop, One Execution Set



The following is an example of a nested loop.

Example 5-16. Nested Loop



In Example 5-16, the LPMARKA bit of loop 0 is set because the SKIPLS instruction can skip over the LPMARKB bit of loop 0.

The assembler sets the appropriate LPMARK bits, adding a prefix word with the loopstart or loopend information, if necessary. In disassembly, these LPMARK bits (if used) appear preceding the normal disassembled mnemonics of the set.

5.4.7 Loop Timing

If the loop starting address is not aligned (meaning that the first execution set is spread over two fetch sets), one stall cycle is added to the loop execution on each iteration of the loop. In every other case, no stall cycles are added to the loop execution time. A loop may be aligned with the assembler directive `FALIGN`, which when placed just before the `LOOPSTART`, will cause the assembler to insert `NOP` instructions in order to align the first execution set of the loop.

At the end of a long loop having `LCn` greater than one, decoding a `LPMARKB` bit does not cause a stall. Whereas decoding a `LPMARKA` bit in the same situation, adds the change-of-flow stall cycles.

5.5 Stack Support

Multitasking creates the impression that the DSP is executing several tasks concurrently, when in reality it is only executing a single task at any given time. The SC140 core has many features that help software designers implement a software stack, and more efficiently support a multitasking real time operating system (RTOS). These features include:

- Two stack pointers: one for the normal stack (NSP) and one for the exception stack (ESP), only one of which is active at a time (referenced as SP)
- Separate user/normal and exception working modes
- Push and pop instructions
- Stack-oriented addressing modes

5.5.1 SC140 Single Stack Memory Use

In a single stack pointer system, each task stack must allocate memory for RTOS function calls and interrupts. Thus, extra memory is allocated on each task stack as shown in Figure 5-6.

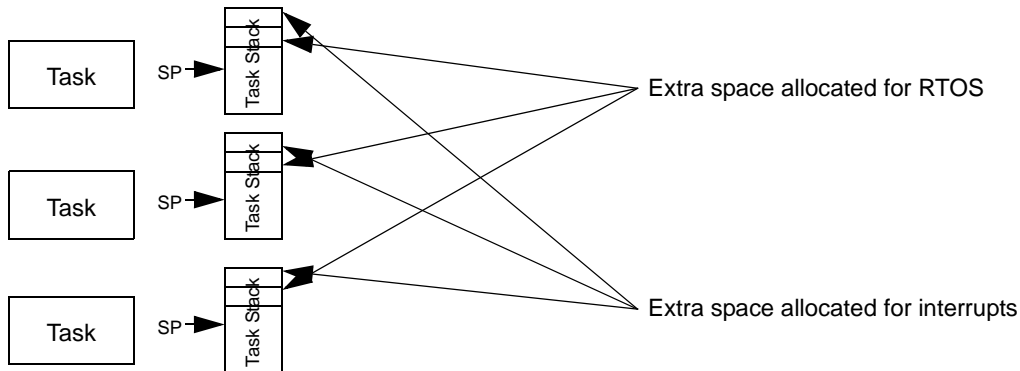


Figure 5-6. SC140 Memory Use with a Single Stack Pointer

The memory space for interrupts is replicated on each task stack since any task can be interrupted. The interrupt functions can use the stack for subroutines, local variables, and so on. So each task stack must be increased by the size of the maximal interrupt memory use.

Memory space is required for interrupts because any task may be active when an interrupt occurs. The ISR pushes registers on the current stack and may also allocate local variables on the current stack. Since it is not known which task is being executed when an interrupt occurs, each task stack must be increased by the maximum ISR memory use. In both situations, the memory is used only once, but it is allocated in more than one location. RTOS functions return without switching tasks. In addition, RTOS calls are not preemptable, although they are interruptible. Interrupts have the same behavior as RTOS functions in that they return without switching tasks.

5.5.2 SC140 Dual Stack Memory Use

The solution to excessive stack memory use is to separate tasks from the RTOS and interrupts. This is done by using the user/normal and exception working modes. The programming model has two stack pointers: NSP and ESP. The NSP is used by tasks when the core is in the user or normal working modes. The ESP is used in the exception working mode by the RTOS and interrupts. Since the RTOS and interrupts have their own stack pointer, memory for the RTOS and interrupts can be allocated separately. Thus, the RTOS and interrupt code can be modified independently of the tasks.

Figure 5-7 shows the stack structure.

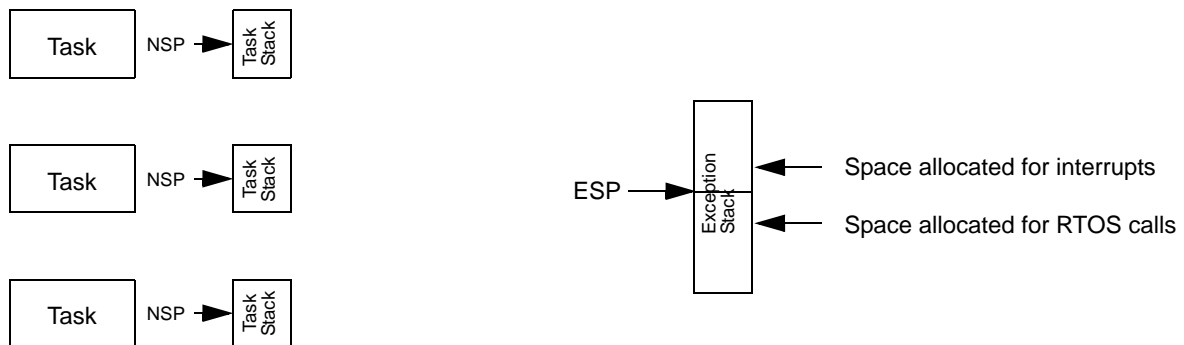


Figure 5-7. SC140 Memory Use with Dual Stack Pointers

The core uses the exception working mode whenever it is processing an exception. When an exception occurs, the core switches to the ESP, saves the PC and SR, and uses the exception stack later for saving registers and allocating local variables and subroutine calls.

RTOS calls are performed by executing a TRAP instructions, which generates a software interrupt. Since the processor is now in exception working mode, all stack memory use is on the exception stack.

As the core enters the exception working mode, the RTOS usually needs to save the current context by storing registers other than the SR and PC in the normal stack. For this purpose, specialized push and pop instructions (PUSHN/POPN) are provided that always access the normal stack, regardless of the mode.

5.5.3 Stack Support Instructions

The core provides push and pop instructions that reference the active stack pointer (NSP or ESP). Table 5-11 describes these instructions.

Table 5-11. Stack Push/Pop Instructions

Instruction	Description
POP	Pre-decrement the stack by eight and restore one 32-bit register
POPEN	Same as POP, but using the NSP regardless of the working mode
PUSH	Push a single 32-bit register onto the active stack and increment the pointer by eight
PUSHN	Same as PUSH, but using the NSP regardless of the working mode

In addition, the stack can be accessed with move or bit mask instructions that use short and word displacement addressing with the stack pointer as a base pointer. However, these instructions do not change the value of the stack pointer. Generally, the stack pointer points to the next unoccupied location.

While using the pop/push instructions, all SC140 registers are viewed as two separate banks, an even register file bank and an odd register file bank (as shown in Table 5-12).

Table 5-12. Even and Odd Registers

Even Register (De) File	Odd Register (Do) File
D0 D2 D4 D6 D8 D10 D12 D14 D0.e D2.e D4.e D6.e D8.e D10.e D12.e D14.e D0.e:D1.e D2.e:D3.e D4.e:D5.e D6.e:D7.e D8.e:D9.e D10.e:D11.e D12.e:D13.e D14.e:D15.e R0 R2 R4 R6 R8 R10 R12 R14 B0 B2 B4 B6 N0 N2 M0 M2 SA0 SA1 SA2 SA3	D1 D3 D5 D7 D9 D11 D13 D15 D1.e D3.e D5.e D7.e D9.e D11.e D13.e D15.e R1 R3 R5 R7 R9 R11 R13 R15 B1 B3 B5 B7 N1 N3 M1 M3 LC0 LC1 LC2 LC3 VBA SR MCTL

Up to two push instructions are supported in a single execution set. If two push instructions are included in a single execution set, one push instruction must use an even register operand, and the other push instruction must use an odd register operand. A push instruction always pushes one 32-bit register into the stack. Any execution set that includes one or two push instructions increments the stack pointer by eight. In the case of a single push, a single operand is written to the memory while the adjacent memory location remains unchanged.

Table 5-13 describes the stack memory map while performing a single or a dual push access.

Table 5-13. Stack Memory Map

Type	Memory Location X+4	Memory Location X
Single push - even register	Unused	Even operand
Single push - odd register	Odd operand	Unused
Dual push	Odd operand	Even operand

Up to two pop instructions are supported in a single execution set. If two pop instructions are included in a single execution set, one pop instruction must use an even register operand and the other pop instruction must use an odd register operand.

An execution set that includes one or two pop instructions restores D_e from $SP-8$ and/or D_o from $SP-4$ (See Table 5-12 for the definition of D_e and D_o). The execution set decrements the original stack pointer by eight as specified by the operands. Note that if the stack is popped with one register only, the data from the other pushed register may be lost.

Pushing and popping the data extension register ($D_{x.e} + L_x$ tag bit) are unique operations. It is possible to push two extensions that are coupled together to form a single operand, or to push a single extension. The single extensions are divided between the even and odd tables. In both cases, the push operation occupies 32 bits. For more information, see Table 5-12, as well as the PUSH and POP instructions in Appendix A.

For correct operation, the stack should be popped in reverse order with exactly the same register pairing as it was pushed. When dual push instructions are used in an execution set, the corresponding pop instructions should be dual. The pop operands should match the corresponding push operands.

In addition to the push and pop instructions, the stack can be accessed directly with move or bit-mask instructions. The available addressing modes are shown in Table 5-14. The two addressing modes differ in the instruction word count. Note that the user cannot use addressing modes that update SP during the access, but only short or word displacement addressing modes that leave the SP unchanged.

Table 5-14. Stack Move Instructions

Addressing Mode	Description
(SP - xx)	Subtract offset by a shifted unsigned 5-bit or 6-bit immediate value. The SP remains unchanged.
(SP + xxxx)	Add a signed 15-bit immediate offset. The SP remains unchanged.

5.5.4 Shadow Stack Pointer Registers

The stack normally grows by incrementing SP and shrinks by decrementing SP. Both stack pointers have shadow registers that contain a decremented value of the stack pointers. When the shadow register is not valid, the pop instruction is executed in two cycles where the first cycle is used to decrement the stack pointer. When the shadow register is valid, the pop instruction is executed in only one cycle.

When an NSP or ESP is written (by TFRA), then its shadow register automatically becomes invalid. In this situation, the first pop instruction takes an additional cycle. When a push/pop instruction is executed, then the shadow register of the active NSP or ESP becomes valid.

5.5.5 Fast Return from Subroutines

The SC140 supports a mechanism for speeding up the execution of the return from subroutine (RTS) instruction, using a return address stack (RAS) register. The RAS is updated with the return address during the execution of a JSR or BSR instruction.

Normal execution of an RTS takes five to six execution cycles. If the routine performing the RTS is a leaf routine (meaning that no other RTS has been executed between the jump to this subroutine and the execution of the RTS), then RTS executes in three cycles. Upon RTS, the RAS is invalidated until the next JSR or BSR instruction.

The user is not allowed to create a situation where upon using RTS, the RAS is valid but the return address does not match the one that is stored in the stack. This situation may occur if the user explicitly changed the return address in the stack. See Rule J.4 in [Chapter 7, “Programming Rules.”](#)

The RTSTK instruction can be used to bypass the special logic that implements this fast RTS mechanism. This instruction retrieves the return address from the stack also when the RAS is valid. RTSTK is typically used when the return address from subroutine is explicitly changed in the stack.



5.6 Working Modes

The working mode is determined by the EXP bit in the Status Register (SR), as shown in the table below:

Table 5-15. Working Modes

Working Mode	EXP bit	Active SP
Normal	0	NSP
Exception	1	ESP

The SC140 can operate in one of two working modes.

Normal mode - Typically intended for task-related services. Works with NSP as the stack pointer.

Exception mode - Typically intended for RTOS kernels, exception routines and peripheral device drivers. Works with the ESP as the stack pointer.

5.6.1 Normal Working Mode

This mode uses the Normal Stack Pointer (NSP). This mode is intended for application tasks that were each allocated a distinct stack area. It could also be used for RTOS services that are tightly related to a specific task, and hence would work more efficiently if they could reference the NSP with SP. Example for such tasks are memory allocation and management tasks, or RTOS messaging services that need observability into the task data structures.

The core stays in this mode unless:

- An exception is encountered, as described in
- A hardware reset occurs, as described in [Section 5.7.4, “Reset Processing State.”](#)
- An RTE-like instruction (RTE/D) is issued,.

5.6.2 Exception Working Mode

This mode uses the Exception Stack Pointer (ESP). It is intended for the RTOS kernel, interrupt service routines, peripheral device drivers, etc. Also, application code for single stack systems runs in this mode.

The SC140 core can enter the exception working mode in any of the following ways:

- An external hardware interrupt request is issued to the core, for example by an off-chip device or an on-chip peripheral.
- A software exception request (TRAP) is issued by the program itself.
- An internal exception such as an illegal opcode, illegal execution set, or DALU overflow occurs.
- A debug exception request is issued by the EOnCE.

This is the default state of the core after exiting the reset state. Refer to [Section 5.8, “Exception Processing,”](#) for a detailed description of the different exception types, and of the way an exception is serviced.

5.6.3 Typical Working Mode Usage Scenarios

The core changes its working mode in different ways, depending on the protection and stack paradigm in use. The sections immediately below illustrate two common task management paradigms supported by the SC140 core, that may be used by an RTOS: Dual-stack, and Single-stack

The terms “single” or “dual” stack refer to the number of stack pointer registers that are used by the system. When using a dual stack-pointer configuration, the RTOS can implement what is termed a multi-stack system. In such configurations, the stack pointer allocated for user tasks can be changed at each task switch to point to a different memory location for each task.

5.6.3.1 Dual-stack RTOS

Figure 5-8 illustrates the working mode transitions for dual-stack operating systems.

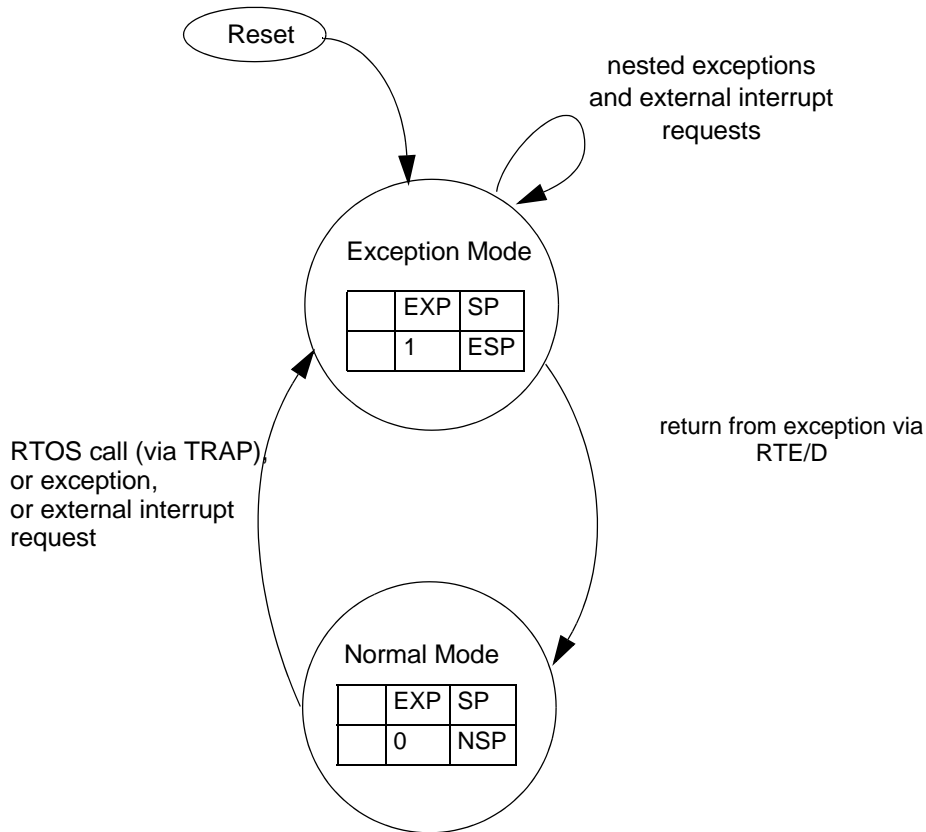


Figure 5-8. Working mode Transitions - Unprotected Dual-stack RTOS

The dual-stack operating system kernel executes in the Exception working mode. User task context is initialized while in the Exception working mode. User task invocation occurs when an RTE/D instruction is executed that restores EXP=0 in the SR. The user task executes in the Normal working mode until it requests operating system services using a TRAP instruction, or an exception or external interrupt request occurs. When the working mode changes from Normal to Exception mode, EXP is set by the core, and the previous SR is pushed on the exception stack.

5.6.3.2 Single-stack RTOS

Figure 5-9 illustrates state transitions for a single-stack-based operating system.

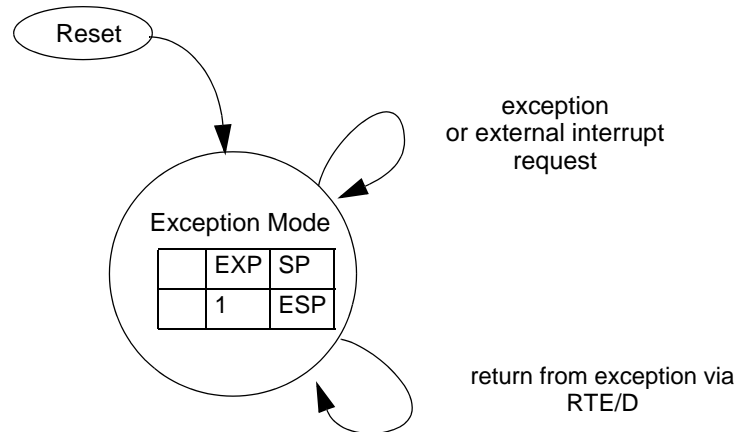


Figure 5-9. Working mode Transitions - Unprotected Single-stack RTOS

Existing single-stack operating systems operate exclusively in the Exception working mode. The EXP bit is always set, making ESP the active SP for all operating system and user tasks. Transfer of control between the operating system and user tasks is typically made via change-of-flow (COF) instructions (JSR, RTS/D and RTSTK/D). If an exception is taken, the exception service routine returns control via RTE/D.

5.6.4 Working Mode Transitions

5.6.4.1 From Exception to Normal mode

The core leaves the Exception mode and enters the Normal mode by either explicitly changing the EXP bit in the SR (for example with the instruction: `BMCLR #4, SR.H`), or executing a Return From Exception (RTE/D) instruction. If this transition is to be taken to a new task that is not on the stack, the stack must be pre-loaded with suitable values before performing the RTE. The following explicit actions must be taken:

- The address of the first VLES for the task must be stored in the memory location representing the PC on the active (ESP) stack.
- The memory location representing SR on the active (ESP) stack must be set with EXP=0.
- An RTE/D instruction restores the PC and SR from the active (ESP) stack.

From the core's point of view, the difference between Exception and Normal modes is only the identity of the active stack pointer (NSP vs. ESP).

5.6.4.2 From Normal to Exception mode

The core leaves the Normal mode and enters the Exception mode upon taking an exception, either via a TRAP instructions, or via an imprecise exception or interrupt request. It can also enter the Exception mode by explicitly setting the EXP in the SR (directly using an instruction or indirectly using RTE where in the restored SR EXP=1). Upon an exception, the following implicit actions occur:

- The EXP bit in the SR is set(if not already), thereby enabling the Exception Stack Pointer (ESP) as the active SP.
- The PC and previous SR are pushed on the active (ESP) stack.
- The PC jumps to the Vector Base Address (VBA) + Exception Offset Address. For example, executing a TRAP instruction causes the core to enter an Exception state and begin executing instructions at VBA + 0x00, since the TRAP instruction has an exception offset address of 0x00.

If choosing to prepare the return values on the stack explicitly to perform this transition, the programmer should be aware that in Normal mode RTE uses the active stack (NSP).



5.7 Processing States

The SC140 core is always in one of the five processing states:

- Execution
- Debug
- Reset
- Wait
- Stop

These states are described in the sections that follow. In some states, the operation of peripherals and other blocks is affected.

Note: The descriptions of the change in operation given here may be different for certain products that utilize a SC140 core. Consult the product-specific manuals for details of actions in each processing state.

5.7.1 Processing State Change Instructions

Processing state changes can be initiated by hardware or software means. Table 5-16 lists the instructions that can initiate a processing state change.

Table 5-16. Processing State Change Instructions

Instruction	Description
DEBUG	Enter debug state
DEBUGEV	Signal a debug event
STOP	Stop processing (lowest power stand-by)
WAIT	Wait for interrupt (low power stand-by)

5.7.2 Processing State Transitions

The transitions between the states are summarized in the following figure.

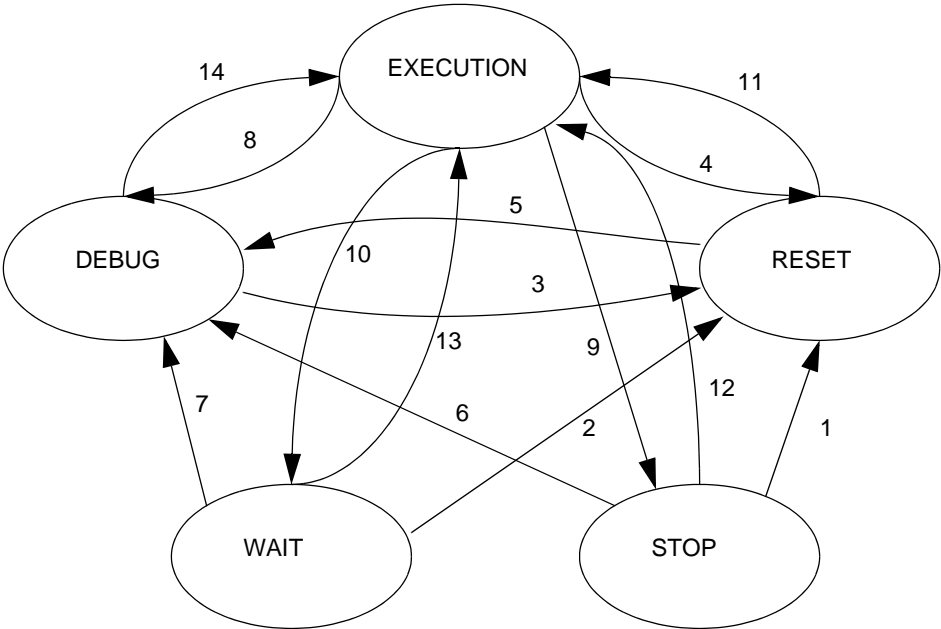


Figure 5-10. Core State Diagram

Table 5-17 describes the processing state transitions shown in Figure 5-10.

Table 5-17. Processing State Transitions

Processing State Transitions	Description
1, 2, 3, 4	Assertion of one of the core hardware reset input signals.
5	De-assertion of reset if EE0 or a JTAG debug command is asserted during reset
6, 7	Entering debug state through an external request (JTAG, EE pin or system input).
8	Entering debug state through execution of debug/debugev, or occurrence of EOnCE events as programmed in the activated EOnCE, or an external request ((JTAG, EE pin or system input)
9	Executing the STOP instruction.
10	Executing the WAIT instruction.
11	De-assertion of the reset signal, assuming EE0 not asserted.
12	Assertion of one of the exit from STOP signals.
13	Assertion of an unmasked interrupt request.
14	Programming change to the EOnCE control registers through the JTAG.

5.7.3 Execution State

The execution state is where instructions are repeatedly fetched and executed. All software runs in the execution state.

5.7.4 Reset Processing State

The reset processing state is entered when an external core hardware reset occurs. Upon entering the reset state, the following registers are updated with their reset values:

- SR
- EMR
- VBA
- MCTL

Refer to [Chapter 3, “Control Registers,”](#) for details on the reset of the various bits of the above registers.

The core remains in the reset state until the end of hardware reset. Upon leaving the reset state, the core enters the exception working mode, as part of the execution state, and program execution begins at a derivative-dependent program memory address.

5.7.5 Debug State

The debug state is a special core processing state in which the pipeline is stalled and waits for user commands from the JTAG or EOnCE. The core can enter the Debug state in the following cases:

- JTAG issues a debug request is asserted, in all states.
- The EE0 EOnCE signal is asserted during reset.
- The EE0 EOnCE signal is asserted anytime, if programmed as a debug request input in the EE_CTRL register.
- An EOnCE Debug state event occurs.
- A DEBUG instruction is executed, if the SDD in EMR is not set, and the EOnCE is programmed so that it will generate a Debug state event.
- A DEBUGEV instruction is executed, and the EOnCE is programmed so that it will generate a Debug state event.

If EE0 or the JTAG debug request are asserted during reset and continue to be asserted when reset is de-asserted, the core enters the debug state without executing any instruction.

The debug state is exited by setting the exit bit in the EOnCE command register by JTAG. Refer to [Chapter 4, “Emulation and Debug \(EOnCE\),”](#) for a detailed description of the user commands in the debug state.

5.7.6 Wait Processing State

The wait processing state is a low-power consumption state entered by the execution of the WAIT instruction. After a system-specific delay of some cycles from the issue of the WAIT instruction, the core’s global clock is turned off. Peripherals can continue to operate, but all internal processing is halted until one of the following actions occurs:

- An interrupt, with enabled priority, is issued¹.
- A non-maskable interrupt (NMI) request is issued.
- A low-level is applied to the RESET signal (RESET asserted).
- The JTAG issues a debug request.
- The EOnCE EE0 signal (programmed as a debug request input) is asserted.

Debug request from the EOnCE may also exit from Wait Processing State, if it occur a few cycles after the WAIT instruction execution (exact time may vary according to the specific clock scheme implemented).

If an exit from the Wait Processing State is caused by assertion of the EE0 signal or a debug request, the core either enters the debug state immediately, or the debug exception is serviced according to the EOnCE configuration. Refer to [Chapter 4, “Emulation and Debug \(EOnCE\),”](#) , for further details.

If the Wait Processing State is exited by assertion of the RESET signal, the core enters the reset processing state.

Table 5-18 describes exit from Wait Process State, due to interrupt and NMI, under various core conditions.

1. i.e. IPL of the interrupt is greater than the core IPL, as determined by bits I2-I0 of the SR. See Table 5-18 for more information.

Table 5-18. Exit Wait Processing State due to an Interrupt or NMI

Interrupt Request	Disable Interrupts (DI)	Disable NMI (NMID)	Wait Process
Maskable Request with IPL > core IPL as determined by the I2–I0 bits of the SR	Clear (interrupts enabled)	Clear or set	Exit the wait processing state. Jump to the Interrupt Service Routine (ISR).
Maskable Request with IPL > core IPL as determined by the I2–I0 bits of the SR	Set (interrupts disabled)	Clear or set	Exit the wait processing state. Enter execution state and continue program execution, following the WAIT instruction. No jump to the ISR.
Maskable Request with IPL ≤ core IPL as determined by the I2–I0 bits of the SR	Clear or set	Clear or set	Remain in the wait processing state.
Non-maskable request	Clear or set	Clear	Exit the wait processing state. Jump to the ISR.
Non-maskable request	Clear or set	Set	Exit the wait processing state. Enter execution state and continue program execution, following the WAIT instruction. No jump to the ISR.

5.7.7 Stop Processing State

The stop processing state is the lowest power consumption state and is entered by the execution of the STOP instruction. After the STOP instruction has been issued, it takes a system-specific number of clock cycles to enter the stop state and turn off the global clocks to the entire core and peripherals. The core exits from the stop processing state when one of the following occurs:

- A dedicated core “wake from stop” input signal is asserted.
- The RESET signal is asserted.
- The JTAG controller issues a debug request.
- The EE0 signal (programmed as a debug request input) is asserted.

Debug request from the EOnCE may also exit from Stop Processing State, if it occurs few cycles after the STOP instruction execution (exact time may vary according to the specific clock scheme implemented).

If an exit from the stop processing state is caused by assertion of the EE0 signal or a debug request, the core either enters the debug state immediately, or the debug exception is serviced according to the EOnCE configuration. Refer to the EOnCE Reference Manual, for further details.

If the Stop Processing State is exited by assertion of the RESET signal, the core enters the reset processing state.

If the stop processing state is exited during the assertion of an external interrupt request, the core enters the exception mode and services the highest priority pending interrupt. If no interrupt is pending, the core enters the execution state and executes the instruction following the STOP instruction that caused the entry into the stop state.



5.8 Exception Processing

Exceptions are events that interfere with the normal operation of the core and the system in which it works. The Exception working mode was designed to deal with situations such as these. In general, the prioritizing and arbitrating between all the exception sources is performed in the programmable interrupt controller (PIC), which is not part of the SC140 core. This section describes the exception handling after the PIC has determined which interrupt request is issued to the core.

A distinction is made between the terms “exception” and “interrupt” in this section. “Exception” is used as a general term for all the cases that interfere with normal program execution, whether generated by hardware or software, internal or external. “Interrupt” is used only for external (off-core) hardware interrupt sources.

There are three categories of exceptions as listed below:

- **Internal Exceptions** — These have the highest priority, ranging from 0 to 3, with 0 the highest. Each of these three, TRAP, ILLEGAL, and DEBUG, has a separate priority and a separate offset address vector. The offset address vector forms part of the address to which the program jumps to perform a particular routine in response to the exception.
- **Non-Maskable External Interrupts** — These have the next highest priority level, four. A non-maskable external interrupt is driven from the external interrupt controller. Its offset address vector is either the AUTO-VEC (0x180) or the value on the 6-bit Interrupt Offset bus.
- **Maskable External Interrupts** — These have the lowest priority level, five (in comparison to the above exceptions). Each occurrence of a maskable external interrupt has an interrupt priority level (IPL) associated with it, driven on the IPL bus. This IPL value is compared to the internal masking threshold defined in the SR. If the IPL exceeds the threshold, it can be serviced. The offset address value is either the AUTO-VEC (0x1c0) or the value on the 6-bit Interrupt Offset bus.

Note that two types of priority terms are in use here. One is the priority among the three major types above, including the four levels in type 1. The second is the interrupt priority level, from 0 to 7, which only applies within the maskable external interrupts. The first priority type, with values from 0 to 5, determines which exception is to be taken if two or more exceptions are pending on the same clock cycle. The second priority type, the interrupt priority level, determines whether a maskable external interrupt is taken or not.

Figure 5-11 below depicts the core interface to an external interrupt controller.

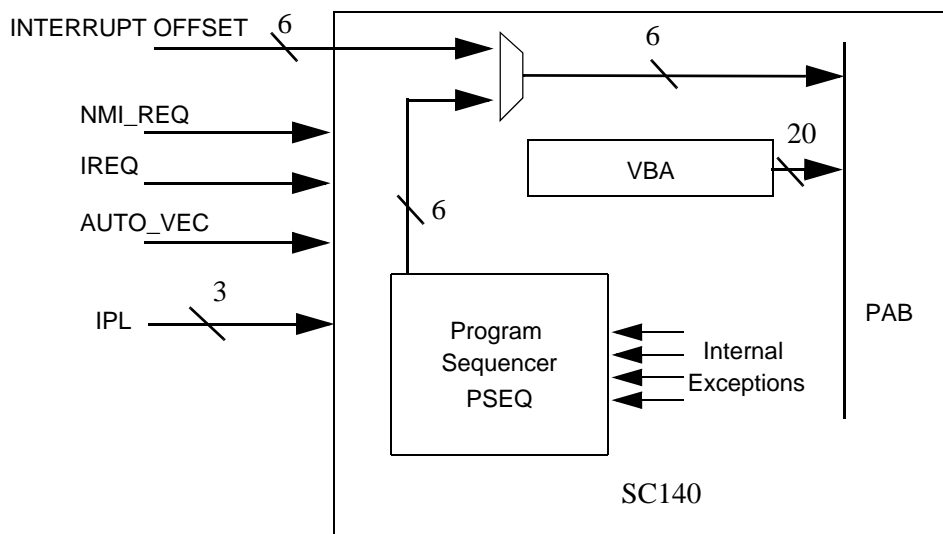


Figure 5-11. Core-PIC Interface

The interface signals (inputs to the core from an external interrupt controller) are described in the following list.

- **Maskable Interrupt Request Signal (IREQ)** — Asserted to inform the core of a pending maskable interrupt request.
- **Interrupt Request Priority Level (IPL)** — This 3-bit bus defines the priority level of the maskable interrupt request. If this value exceeds the value encoded in the SR bits I[2:0], the interrupt can be serviced. Otherwise, it is masked.
- **Non-maskable Interrupt Request (NMI_REQ)** — Assertion of this signal initiates an interrupt independent of the value on the IPL bus or the SR priority level. This input can be inhibited by the non-masked interrupt disable (NMID) bit in the Exception and Mode Register (EMR).
- **Enable Auto-vector (AUTO_VEC)** — This signal selects the source for the offset part of the interrupt vector address: either the default offset or the value driven by the user on the 8-bit INTERRUPT OFFSET bus. This selection affects both maskable and non-maskable interrupts.
- **INTERRUPT OFFSET** — This 6-bit value can be the offset address applied to the interrupt vector address table. It is selected if the AUTO_VEC (above) does not select the default offset. The complete address of the interrupt vector is defined by a number of fields. For more details, see [Section 5.8.1.2, “Programming Exception Routine Addresses.”](#)

The following list outlines how exceptions are processed by the SC140:

1. The hardware interrupt request is synchronized with the core clock. The interrupt pending request signal for that particular hardware interrupt is set. An internal exception (such as an illegal instruction) is processed in the PSEQ internally as a non-maskable exception request. An exception source can have only one exception pending at any given time.
2. The PSEQ in the core automatically ignores any interrupt request with an IPL lower than or equal to the interrupt mask level in the SR. NMIs and internal exceptions are serviced regardless of the current IPL.



3. The PSEQ services an exception request when ready, typically in five cycles. It may postpone an exception while a change-of-flow is executing (up to 16 cycles latency). After fetching an exception service routine base address, the core enters the exception working mode. The next PC value (namely, the address of the execution set where execution should be resumed upon the return from the exception) is pushed onto the exception stack together with the SR. The instruction set at the exception vector address associated with the selected exception is fetched. This address is formed as follows:
 - Bits 31:12 from the Vector Base Address Register (VBA)
 - Bits 11:6 from either the exception and interrupt address offset table (Table 5-19) or the external Interrupt Offset Bus as enabled by the AUTO_VEC signal and an external interrupt. External interrupts with a default vector address (indicated by the AUTO_VEC signal) refer to the AUTO-NMI and AUTO-IR vectors in Table 5-19.
 - Bits 5:0 of an exception vector base address are always zero, allowing 16 bytes at each vector.
 - The first three execution sets from the fetched exception vector enter execution. Only then can a new exception request be serviced.

5.8.1 Interrupt Vector Address

5.8.1.1 Vector Base Address Register

The Vector Base Address Register (VBA) is a 32-bit register with the lower 12 bits always zero. The upper twenty bits [31:12] are automatically used to form the base address (bits [31:12] of the exception vector address). The upper twenty bits of VBA are initialized at reset with a derivative-dependent address pointing to the initial Vector Address Table. After reset, VBA may be programmed to relocate the Vector Address Table anywhere in memory. Care must be taken that interrupts are disabled (either in the core or in the system) when changing the VBA register, otherwise an interrupt serviced during the change process may use either the old or new VBA value.

5.8.1.2 Programming Exception Routine Addresses

Each exception vector address is formed from a base address and an offset. The base address bits [31:12] come from the VBA register. Bits 11:6 of the offset are from either the exception vector address table (Table 5-19) or the external Interrupt Offset Bus as enabled by the AUTO_VEC signal and an external interrupt. External interrupts with a default vector address (indicated by the AUTO_VEC signal) refer to the AUTO-NMI and AUTO-IR vectors in Table 5-19. Bits [5:0] are zero since the distance between two exception vectors is 16 bytes (one full execution set). There are 64 possible exception vector locations in the table.

Table 5-19 shows the exception vector address offsets. The last row in the table is for offsets from 0x200 to 0xFC0. These can be accessed by either the non-maskable interrupt or the (maskable) external interrupt, since the user-driven Interrupt Offset bus determines this address for either type.

Table 5-19. Exception Vector Address Table

Exception Address Offset	Priority (0 - highest)	Type	Description
0x00	0	TRAP	TRAP instruction
0x40	-	Reserved	
0x80	1	ILLEGAL	ILLEGAL instruction, and illegal instruction set
0xC0	2	DEBUG	DEBUG exception from EOnCE
0x100	3	Overflow	DALU overflow
0x140	-	Reserved	
0x180	5	Auto-NMI	NMI default vector
0x1C0	6	Auto-IR	Interrupt default vector
0x200-0xFC0	5	NMI	NMI
		External interrupts	External interrupts

5.8.2 Return From Exception Instructions

Return from exception should be done with dedicated Return from Exception instructions (RTE, RTED, - termed collectively “RTE-like” instructions). These instructions pop from the active stack two values: The return PC, from which execution resumes, and SR value. The SR value sets (among other things) the working mode of the core. The RTE-like instructions are used to automatically enable resuming the task that was interrupted (by restoring the next PC to be executed and the SR, including the working mode). It could also be manipulated by the RTOS to change the task that is restored.

For maximum efficiency on return from an exception, the SC140 instruction set provides a delayed return from the exception instruction. This takes five or six cycles to execute, but allows the usage of some of these cycles to execute instructions. Refer to [Appendix A, “SC140 DSP Core Instruction Set,”](#) for details on return-from-exception usage in the RTE and RTED instructions.

The RTE/D instructions do not affect the shadow return address of subroutines, see [Section 5.5.5, “Fast Return from Subroutines.”](#) In this way, the interrupts interfere less with the interrupted task, allowing it to continue and enjoy the reduced cycle count when performing an RTS, if applicable. However, if during the ISR, the task was switched, the RAS may hold content from the previous task that may corrupt the execution of the restored task. In such a case, the user should clear the RAS by performing, in the ISR that switches the tasks, a dummy jump to subroutine (JSR) and return (RTS).



5.8.3 Maskable Interrupts

5.8.3.1 Interrupt Priority Level

An external maskable interrupt is given a request IPL (between 1 and 7) by driving a 3-bit input along with the request. The core IPL is held in the I2–I0 bits of the SR. Only interrupts with a request IPL greater than the core IPL are serviced. Refer to [Section 3.1.1, “Status Register \(SR\),”](#) on page 3-1, for further information.

Upon entry to execution, the core priority level is set to be equal to the interrupt priority level of the serviced interrupt. This prevents interrupts with lower and equal priority level from entering execution. In particular it prevents the same interrupt, which usually is still asserted, from entering execution again. In the Interrupt Service Routine the user should typically include an access to the interrupt source to de-assert the interrupt request, after which the previous interrupt priority level could be lowered, either explicitly or by executing the RTE instruction which restores the original SR.

5.8.3.2 Controlling All Interrupt Sources

All maskable interrupts can be disabled with the DI instruction, which sets the DI bit in the SR. No interrupts are serviced after the DI instruction is executed. As a result, the code following the DI instruction does not need to take into account any possible pipeline effects caused by interrupts. Non-maskable exceptions are not blocked by the DI instruction. The EI instruction clears the DI bit in the SR, thus enables all interrupts that are not masked by the IPL bits. The DI and EI instructions do not affect the IPL bits.

5.8.4 Non-Maskable Interrupts (NMI)

An NMI request is serviced regardless of the current IPL and DI bit values. The only time an NMI request remains pending is when another NMI is already being serviced. When an NMI service routine enters execution (namely, the NMI vector is fetched), the NMI disable (NMID) bit in the EMR is set. Refer to [Section 3.1.2, “Exception and Mode Register \(EMR\),”](#) for a detailed description of the EMR. While this bit is set, any pending NMI request is not serviced if the core is in normal or exception modes. This bit is cleared by an RTE or RTED instruction (while the core is in normal or exception mode), or by writing 1 to it. It cannot be set by the user.

5.8.5 Internal Exceptions

This section describes exceptions generated by conditions inside the core. The internal exceptions (except the TRAP instruction) are **imprecise**. These exceptions occur asynchronously after detecting the exception condition. Thus, they are unable to identify the precise location of the offending instruction. They are used mostly for diagnostics during program debugging.

In order to aid in the debugging process, a dedicated register in the EOnCE (PC_EXCP) holds the address of the VLES that caused internal, imprecise exceptions (the Illegal and Overflow exceptions). For multiple exception types (illegal instruction, illegal execution set, or DALU overflow), the PC_EXCP register will retrieve the VLES address of the first occurrence of the last exception type. For multiple events of the same type, only the first event will be sampled in PC_EXCP. For more information on the PC_EXCP register, see [Section 4.7.8, “PC of the Exception Execution Set \(PC_EXCP\).”](#)



If two or more exceptions are pending on the same clock cycle, the one with the higher priority (as defined in Table 5-19 on page 5-49) is taken.

Due to the imprecise nature of these exceptions, there may be additional exception events between the first event and its exception service routine. Additional exception events will set their associated EMR bits and be serviced according to their exception priority, not necessarily according to the order of the execution sets that caused them.

5.8.5.1 Illegal Exception

The illegal exception is generated by any of several conditions, described in the following sections. To enable application developers to debug applications and avoid illegal conditions and errors, the SC140 core provides exception bits in the EMR, which are set when an exception is detected. The EMR is described in detail in [Section 3.1.2, “Exception and Mode Register \(EMR\).”](#) the address of the execution set that caused the last Illegal exception is written to the PC_EXCP register of the EOnCE.

5.8.5.1.1 Illegal Instruction

An illegal instruction exception is generated when one or more of the instruction opcodes coming from the program memory do not belong to the SC140 instruction set. This exception can also be generated by the ILLEGAL instruction. To prevent the system from entering a deadlock state whenever there is an illegal instruction, an internal exception request is generated, and the ILIN bit in the EMR is set. The execution flow continues until the exception is serviced. Execution of the original program is undefined.

The ILIN bit does not block subsequent illegal instruction exceptions. Multiple illegal instructions will cause multiple illegal exceptions, regardless of the ILIN state. However, illegal instructions that occur close together may share the same illegal exception. In particular, additional illegal events that occur between the first event and its illegal exception service routine will share the same exception. If an illegal execution set also occurred during this period, the ILST bit in EMR will be set to indicate multiple causes for this illegal exception. If the illegal exception service routine has an illegal instruction, nested illegal exceptions will occur.

5.8.5.1.2 Illegal Execution Set

An illegal execution set exception is generated whenever one of the following execution set grouping rules is violated:

- A maximum of four DALU instructions per set can occupy different modulo four positions within the set.
- A maximum of two AGU instructions per set can occupy different modulo two positions within the set.
- A maximum of two extension words per set can occupy different modulo two positions within the set.
- A maximum of one ISAP instruction is allowed per set.

Whenever an illegal set occurs, an exception request is generated. The ILST bit in the EMR is set, and instruction execution continues until the exception is serviced. Execution of the original program code is undefined after this exception occurs.

The ILST bit does not block subsequent illegal execution set exceptions. Multiple illegal execution sets will cause multiple illegal exceptions, regardless of the ILST state. However, illegal execution sets that occur close together may share the same illegal exception. In particular, additional illegal events that occur between the first event and its illegal exception service routine will share the same exception. If an illegal

instruction also occurred during this period, the ILIN bit in EMR will be set to indicate multiple causes for this illegal exception. If the illegal exception service routine has an illegal execution set, nested illegal exceptions will occur.

5.8.5.2 DALU Overflow

The DALU overflow exception is generated whenever an overflow occurs as a result of a DALU operation. Whenever there is an overflow, an exception is generated and the DOVF bit in the EMR is set, if the exception enable bit OVE in the SR is set. The DOVF bit blocks subsequent DALU overflow exceptions. Once the DOVF bit is set, no additional DALU overflow exceptions will occur until the DOVF bit is cleared. Although useful for algorithm debugging, the overflow exception routine may be unable to take corrective action due to the imprecise nature of this exception.

The PC of the execution set causing the overflow exception is saved in the PC_EXCP register of the EOnCE.

5.8.5.3 TRAP Exception

Immediately after executing a TRAP instruction, the core enters the exception mode. Both the PC and SR values are pushed onto the exception stack. The IPL is set to its maximum value and the exception mode is entered. This exception is **precise**. It occurs immediately after the execution set that contains the TRAP instruction.

The TRAP instruction is typically used for RTOS calls.

5.8.5.4 Debug Exception

A debug exception can be initiated as a result of a debug event, as configured in the EOnCE. It is also possible to configure the DEBUG and DEBUGEV instructions to generate a debug exception. This exception is not precise. Please refer to [Chapter 4, “Emulation and Debug \(EOnCE\),”](#) for further details.

5.8.6 Exception Interface to the Pipeline

When an interrupt request signal is asserted or an internal exception is triggered, the PSEQ finds the first possible time slot to interrupt the current program flow and start servicing the exception. The PSEQ will delay an exception service when the current pipeline state cannot be interrupted safely without damaging the running task. In most cases, only one cycle is consumed for the exception servicing (for pushing the return address and the SR onto the exception software stack). From this point on, the exception handling routine is treated like any normal flow code. The sections that follow describe the exception process.

5.8.6.1 Exception Routine Fetch

When the PSEQ acknowledges an exception request for service, the exception vector address is driven onto the program address bus. The core then enters exception mode, fetching instructions starting at the exception vector address.



5.8.6.2 Exception Mode Execution

An exception mode execution is performed in exactly the same way as a normal program flow. There is no constraint on the length of an exception routine. Table 5-20 shows the flow for the pipeline changing from normal execution to exception execution.

Table 5-20. Exception Pipeline

Operation	Instruction Cycle											
	1	2	3	4	5	6	7	8	9	10	11	12
Pre-fetch	n1	n2	n3	i1	i2							
Fetch		n1	n2	n3	i1	i2						
Decode			n1	n2	push	i1	i2					
Address Generation				n1	n2	push	i1	i2				
Execute					n1	n2	push	i1	i2			
n = normal or user execution set i1, i2 = exception execution sets 1 and 2												

5.8.7 Exception Timing

When an unmasked exception is taken, the core breaks the normal execution flow and adds a cycle. The return PC and SR are pushed onto the stack, and the core then resumes execution at the exception vector address. After the exception request is asserted, the exact point at which the normal execution flow is interrupted is not fixed. It is dependent on the properties of the instructions being executed in the vicinity of the exception request as well as any core stalls that may occur in parallel. For example, a delayed instruction and its delay slot constitute an uninterruptable sequence.

Example 5-17 describes the exception servicing for a simple case that does not include delayed instructions or core stalls, but does include exceptions occurring near change-of-flow instructions. In this example, the JUMP instruction represents all change-of-flow instructions in Table 5-8, excluding delayed instructions and TRAP. It also represents the DI (disable interrupt) instruction.

Example 5-17. Basic Exception Timing

Let ES0 -> ... -> ES4 ... be a sequence of execution sets such that if ES0 is a JUMP instruction, then ES1 is an instruction from the target address (ES0 -> ES1 -> ES2 -> ES3 -> ES4). In addition, assume that an exception request arises on the same cycle that ES0 starts its AGU execution stage.

If (ES1 is not JUMP) and (ES2 is not JUMP):

Then

- The execution set from the target of the exception vector is executed after ES2, and the address of ES3 is pushed as a return address to the stack.
- 1 cycle is added, which is needed to push the return address to the stack.

Else, if (ES1 is not JUMP) and (ES2 is JUMP):



Then

- The execution set from the target of the exception vector is executed after ES1, and the address of ES2 is pushed as a return address to the stack.
- 2 cycles are added.

Else, if (ES1 is JUMP):

Then

- The execution set from the target of the exception vector is executed after ES0, and the address of ES1 is pushed as a return address to the stack.
- 3 cycles are added.

End

Figure 5-12 provides a flow chart for Example 5-17.

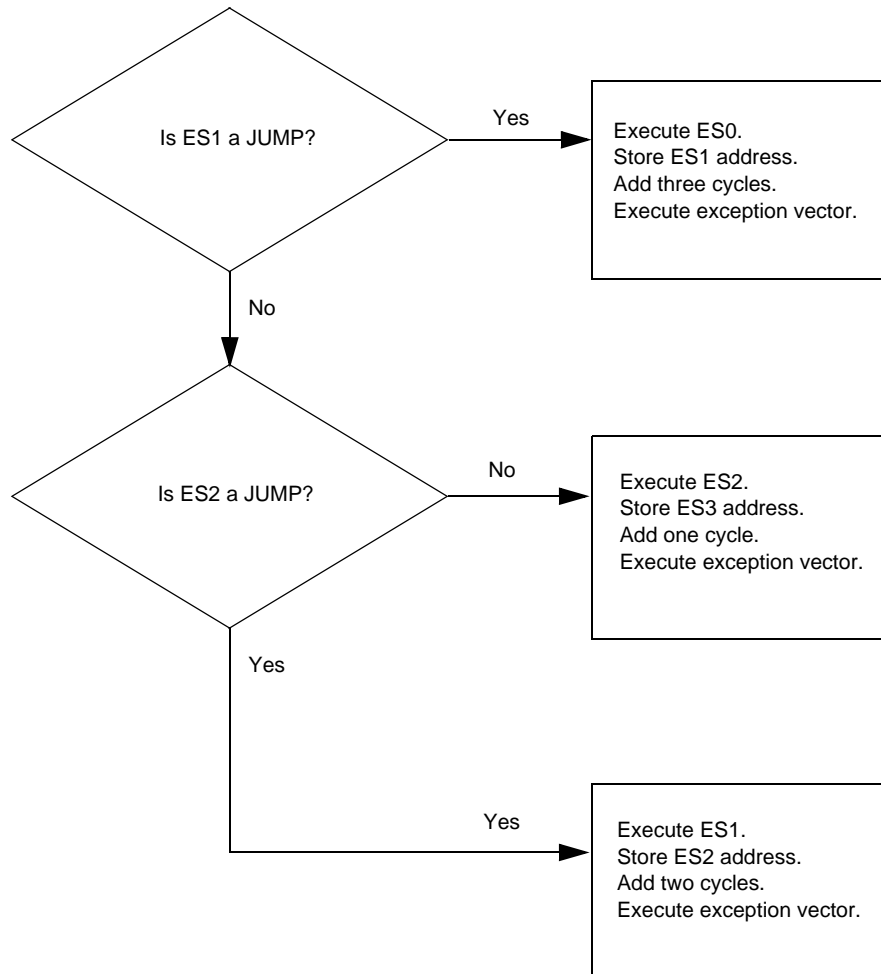


Figure 5-12. Flowchart for Exception Timing

The following pipeline table shows the first case in Example 5-17. ES0 is a JMP with a minimum cycle count of three. ES1 and ES2 are not change-of-flow instructions. And, I1 is the first instruction at the exception vector address. The exception request is initiated in cycle 4.

Table 5-21. Pipeline Example

Operation	Instruction Cycle											
	1	2	3	4	5	6	7	8	9	10	11	12
Pre-fetch	ES0			ES1	ES2		I1					
Fetch		ES0			ES1	ES2		I1				
Decode			ES0			ES1	ES2	push	I1			
Address Generation				ES0			ES1	ES2	push	I1		
Execute					ES0			ES1	ES2	push	I1	

Chapter 6

Instruction Set Accelerator Plug-In

This chapter describes the ISAP capability of the SC140 core, and how to incorporate an ISAP when using the SC140 core as part of a larger system.

6.1 Introduction

An ISAP is an *Instruction Set Accelerator Plug-in* - a unit external to the core that the SC140 core controls using dedicated instructions that are incorporated into the SC140 program. The SC140 has an interface and dedicated encoding space that enables defining and integrating such a unit.

The ISAP enables you to enhance the core instruction set with additional application-specific instructions that are not part of the original SC140 instruction set. For example, ISAPs could enhance the SC140 to support floating-point arithmetic, image processing capabilities, etc.

ISAP support is integral to the SC140 development tools, so the SC140 assembler, simulator and compiler can all be modularly configured to support the specific ISAP(s) definition. This enables the programmer to effectively use ISAP instructions in the source code like any other SC140 instruction (with minor modifications). However, the specific ISAP instruction set and programming rules are specific to each ISAP and are out of the scope of this document.

Single or multiple ISAPs can be connected to a single SC140 core. The normal configuration is of instructions being dispatched to one ISAP at a time.¹

An ISAP normally has its own set of registers, by which it can exchange data with the core or data memory, as well as execute data processing instructions.

1. However, using the SIMD (single instruction, multiple data) approach, more complex configurations that enable dispatching an instruction to more than one ISAP at a time are possible. For details, see the *Integration Guide*.

6.2 ISAP - SC140 Schematic Connection

The ISAP-SC140 connection actually involves an external data memory bank as well. Two connection schemes are shown: SC140 to single ISAP, and SC140 to multiple ISAPs.

6.2.1 Single ISAP

Connection with the ISAP is illustrated in Figure 6-1 below:

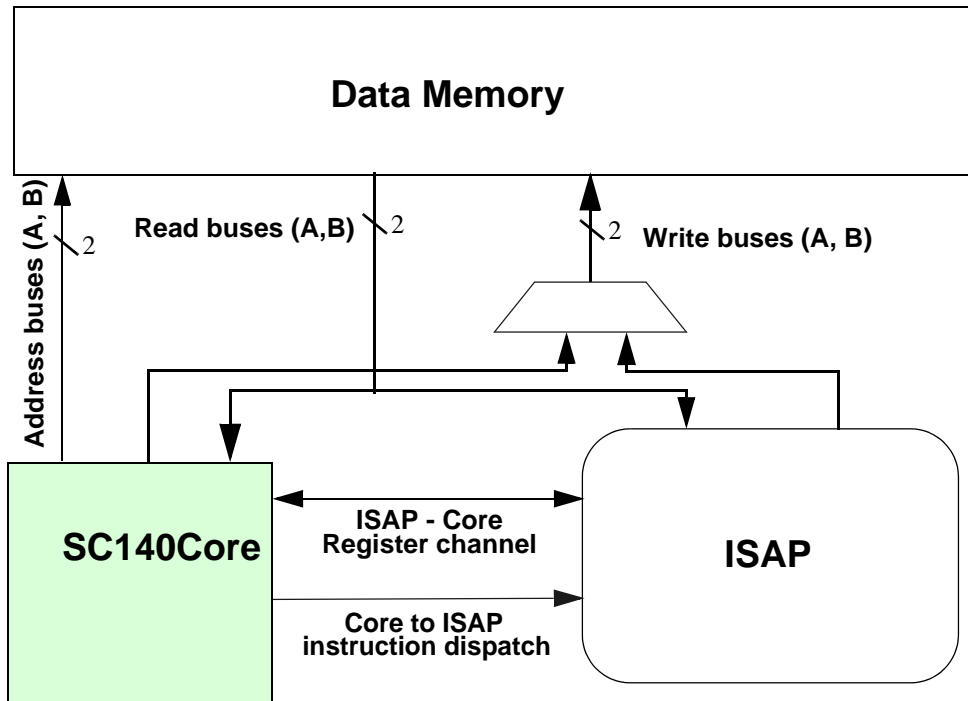


Figure 6-1. Core to Single ISAP Connection Schematic

The ISAP receives instructions from the core via a dedicated ISAP instruction dispatch bus, up to once per execution set.

The ISAP is connected to the data memory via the same two data buses XDBA and XDBB as the core, in both read and write directions. The SC140 core is the only address generating master of the data buses. The ISAP does not send address information to the Data memory, hence the ISAP does not need an AGU. A data access to or from the ISAP requires a parallel core AGU MOVE instruction that generates the access on the address and control lines to the memory. The ISAP then drives or samples the data buses accordingly. The way the ISAP memory access is shared between the core and the ISAP is described in [Section 6.4, “ISAP Memory Access.”](#)

6.2.2 Multiple ISAP

Connection between the core and multiple ISAPs is illustrated in Figure 6-2, below:

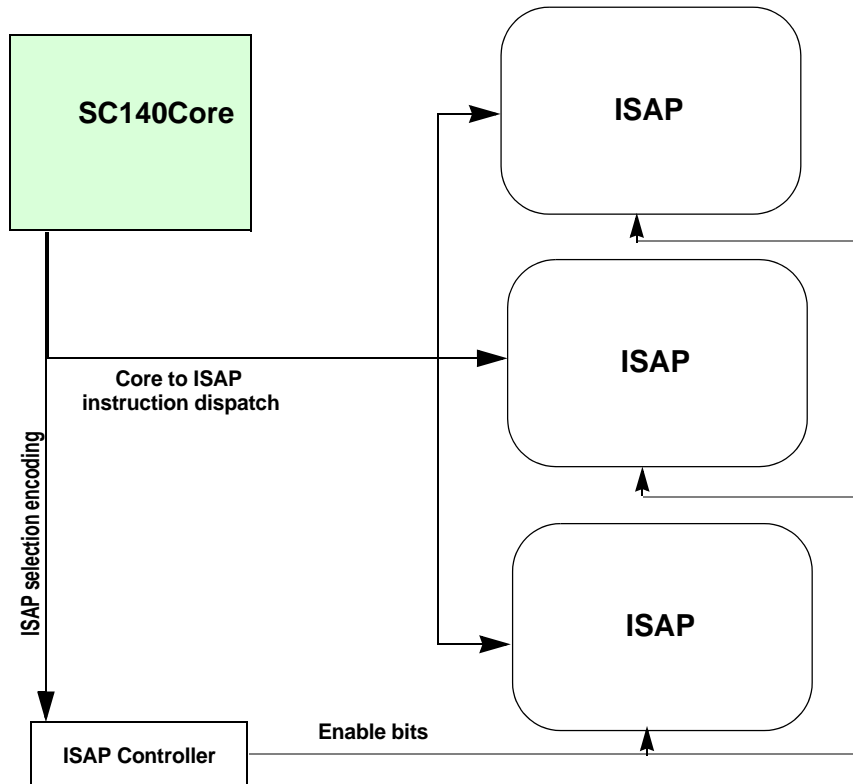


Figure 6-2. Core to Multiple ISAP Connection Schematic

In a multiple ISAP configuration, some of the ISAP instruction encoding bits should be dedicated in advance for encoding the ISAP selection. For each dispatched ISAP instruction, an ISAP controller decodes the ISAP select bits and enables the respective ISAP. The other ISAPs are therefore disabled, for this cycle. The system designer must put these bits in the MSB of the opcode of the ISAP instruction. Further operation is similar to a single ISAP:

The connections of the ISAPs with the data memory are not shown in Figure 6-2. Proper muxing should be implemented according to the same principle as shown in Figure 6-1.



6.3 ISAP instructions and instruction encoding

This section presents an overview of the concept of programming the ISAP from the SC140 assembly code.

The SC140 core can dispatch one *ISAP opcode* per VLES. This opcode uses the 2-word prefix encoding, and is recognized as an ISAP opcode if it is not the first opcode in the VLES. The SC140 core encoding rules allow this situation only with prefix grouping, hence it follows that an ISAP opcode can only appear as part of a prefixed VLES.

The 2-word prefix encoding is shown in [Table 6-1: ISAP Encoding Fields](#) below:

Table 6-1. ISAP Encoding Fields

Binary Encoding	Words	Bits
0011 _____ 101_____	2	25

This encoding allows the ISAP 25 bits as its encoding space. The ISAP architect can freely allocate these bits. The only constraint is that in case the ISAP is intended to be used as part of a multi-ISAP configuration, some of the Most Significant Bits of the 25 bits should be reserved for ISAP selection encoding. The number of such bits that are to be left aside is also the decision of the ISAP architect.

An ISAP opcode can specify one or more *ISAP instructions*. An ISAP could be defined as a VLIW processor, encoding more than one instruction in one ISAP opcode.

ISAP instructions can be of two main kinds:

- *ALU instructions*: ISAP instructions that execute with a pipeline similar to the core DALU, without activation of simultaneous core AGU instructions.
- *Data move instructions*: Instructions that transfer data between the ISAP and its environment (external Data memory or SC140 core). The core has the capability of providing the ISAP with data addressing, so that the ISAP designer need not incorporate an AGU in the ISAP. This requires certain conventions, which are detailed in [Section 6.4, “ISAP Memory Access,”](#) .

ISAP instructions must obey the same VLES semantics as other SC140 instructions (dependency of operands between VLES, etc.). At the instruction level, the syntax should be as close as possible to existing SC140 conventions (source and destination order, data width specifiers, etc.). The individual ISAP instruction syntax should be defined in the particular ISAP specification.

The term “ISAP instruction” can be used both to specify the assembly mnemonic, or to the specific sub-opcode in the ISAP opcode related to the source mnemonic. Differentiation is according to context - source code or assembly level context, or encoding context.

6.4 ISAP Memory Access

The ISAP accesses data via the XDBA and the XDBB busses, in addition to core-ISAP buses for register transfer.

The ISAP should not be designed with an AGU, leaving the addressing duties to the core. This allows the ISAP to enjoy the full addressing and modulo capabilities of the core AGU, and the ISAP design to concentrate on the specific ISAP specialty.

However, this feature requires some assembler support (core and ISAP) when using such instructions. When the ISAP must either write or retrieve data from the data memory via the data busses, the core assembler must create a parallel AGU MOVE instruction in the same VLES as the ISAP instruction.

This method works as follows:

When the assembler encounters an ISAP data move instruction, the core assembler creates an AGU MOVE instruction that will generate the required address to the data memory.

Example 6-1. ISAP memory access

To understand this, look at the following lines of code:

```
core_ins {move_special k0, (r1)}
```

In this example, two parallel instructions are used:

1st - **core_ins** = a generic core instruction

2nd - **{move_special k0, (r1)}** = a fictional ISAP instruction (for illustration purposes only), whose intent is to take the data in k0 (an ISAP register), and place it in the address whose value is found in the core register r1

The core assembler translates the above line of code (at the opcode level) to read, in effect, the following:

```
core_ins move.l d0, (r1) {move_special k0, data bus}
```

In order for this to work, the core must not drive d0 to the memory. The core hardware achieves this in the following way: A memory MOVE instruction that uses a DALU register (D0 to D15) in parallel to any ISAP instruction simply will not drive or sample the D register. In effect, the d0 in this instruction becomes a dummy source which the core therefore ignores, and in our case only the address found in r1 is driven to on the data address bus. In a complementary manner, the ISAP `move_special` instruction **only** drives the data in k0 on the appropriate data bus, and not handle the addressing of the access.

In this manner, the data memory receives data from the ISAP, and the address from the AGU of the core.

There are some ramifications of this method of memory access, that are mentioned in [Section 6.8.4, “Sequencing rules for T bit update.”](#)

The ISAP must support both big and little endian byte order conventions in the same manner as the core supports these conventions (see [Section 2.4.1, “SC140 Endian Support,”](#)).

6.5 ISAP-core register transfers

The ISAP architect can define ISAP instructions that exchange data between core and ISAP registers. These instructions are treated in a similar manner to MOVE instructions. That is, the core assembler translates the ISAP instructions into two equivalent instructions: one core instruction that drives or samples the core register, and an ISAP instruction that samples or drives the ISAP register.

The assembler uses the core instruction “MOVE.L C4<->Db” (See Appendix A, [Appendix , “MOVE.L,”](#) on page A-275). When executed in parallel with an ISAP instruction, the core does not drive or sample the “Db” (DALU) register. This enables the core to perform data transfers between the C4 registers (D, R, N, B, M) and ISAP registers.

Example 6-2. ISAP-Core register transfers

The following line of code,

```
core_ins {move_special d1,k0}
```

That uses these instructions,

1st - **core_ins** = a generic core instruction

2nd - **{move_special d1,k0}** = a fictional ISAP instruction (for illustration purposes only), whose intent is to take the data in d1 (a core register), and place it in the k0 ISAP register.

Is translated by the core assembler (at the opcode level) to read, in effect, the following:

```
core_ins move.l d1,d0 {move_special bus,k0}
```

The core *does not* take the data in d1 and place it in d0, but it takes the data in d1, and drives it on the core-ISAP register bus. In a complementary manner, the ISAP `move_special` instruction should **only** sample the data from the core-ISAP bus, and place it in the k0 ISAP register.

6.6 Immediate Data Transfer to ISAP registers

The ISAP architect can define ISAP instructions that write immediate data to ISAP registers. These instructions are treated in a similar manner like memory moves instructions: the assembler translate the ISAP instructions into two instructions: one core instruction that writes an immediate value to a dummy core register, and an ISAP instruction that samples the data to an ISAP register.

The assembler can use any move-immediate core instruction to a DALU register, for example “MOVE.L #s32,D0” (See [Appendix](#) , “MOVE.L,” on page A-272). When executed in parallel with an ISAP instruction, the core does not sample the data to a DALU register. This enables performing immediate data transfers to ISAP registers.

Example 6-3. ISAP-Core register transfers

The following line of code,

```
core_ins {move_special #$1234,k0}
```

That uses these instructions,

1st - **core_ins** = a generic core instruction

2nd - **{move_special #\$1234,k0}** = a fictional ISAP instruction (for illustration purposes only), whose intent is to take the number \$1234 and place it in the k0 ISAP register.

Is translated by the core assembler (at the opcode level) to read, in effect, the following:

```
core_ins move.l #$1234,d0 {move_special bus,k0}
```

The core *does not* take the number \$1234 and place it in d0, but it takes the number \$1234, and drives it on the core-ISAP register bus. In a complementary manner, the ISAP `move_special` instruction should **only** sample the data from the core-ISAP bus, and place it in the k0 ISAP register.



6.7 Core Assembly Syntax with an ISAP

This section describes aspects of the core assembly syntax, supported by the software tools, that relate to the presence of an ISAP and to ISAP assembly instructions.

6.7.1 Identification of ISAP instructions

ISAP specific commands are included in brackets: {}, for example:

```
add d0,d1,d2 {...ISAP instructions..}
```

The instructions outside the brackets are core instructions, while the instructions inside the brackets are ISAP instructions. The mnemonics used for the ISAP instructions are defined by the ISAP architect, and assembled by an assembler extension module that is modularly linked with the standard core assembler. ISAP instructions can be combined in one bracket pair or could be split in two pairs with different predication, for example:

```
IFT {ISAP instruction} add d0,d1,d2 IFF {ISAP instruction}
```

See more details on predication of ISAP instructions in [Section 6.7.3, “Conditional Execution.”](#)

When using an ISAP, the core assembler should be informed which ISAP is to be used so that it can verify that it is using the correct assembler extension. The core assembler supports two methods to do this: Setting a default ISAP name, and prefixing the ISAP brackets with the ISAP name.

6.7.1.1 Working with One ISAP

When working with a single ISAP, we recommend defining a default ISAP name. An assembly directive is used to do this. The following syntax is used:

```
ISAP_ID_default "ISAP_ID1"
...
{.. ISAP instructions .. }
```

The assembler directive “ISAP_ID_default” defines the default ISAP name (in this case) to be “ISAP_ID1”. The specific ISAP designation is to be determined by the ISAP architect. Until set to another value, the core assembler will assume this ISAP ID string for any ISAP instructions that are not prefixed by an explicit ID string.

This method is preferred when a single ISAP is used in an assembly code section. In this way the overhead of prefixing every ISAP instruction is avoided.

Example 6-4. Single ISAP coding

A VLES that uses an implicit ISAP ID string:

```
nop {tsteq k0} abs d0
```

In this example, three parallel instructions are used:

1st - **nop** = a core instruction

2nd - **{tsteq k0}** = a fictional ISAP instruction (for illustration purposes only), written in the syntax of the appropriate ISAP.

3rd - **abs d0** = a core instruction

The syntax defines that the string between the brackets is sent to the ISAP assembler.

One ISAP in a Single-Line VLES

```
mac d0,d1,d3 {isap_instruction k0,k1,k2 move_special.l k2,(r0)+}
```

In this example, the MAC instruction is executed by the core and the instructions delimited by the brackets are executed by the ISAP.

One ISAP in a Multi-Line VLES

A VLES spanning multiple lines (parallel execution specified by square bracket delimiters and ISAP instructions specified by curly bracket delimiters):

```
1 [ mac d0,d2,d4    mac d1,d3,d5
2 {isap_instruction k0,k1,k2
3   move_special.l k2,(r1)+}
4   move.l (r0)+,r2]
```

In this example, five parallel instruction are executed - three instructions are executed by the core, and two instruction by the ISAP.

line 1: Two MAC instructions executed by the core

line 2: An ISAP instruction with ISAP registers k0,k1 and k2 as arguments

line 3: An ISAP move instruction, storing ISAP register k2 to a memory location pointed by core register r1. The assembler generates an implicit AGU instruction: “`move.l d0,(r1)+`”. As explained in Section 6.4 ISAP Memory Access, the d0 register is a dummy register which is not driven on the memory data bus, rather the ISAP stores its k2 register by driving the k2 register data on the data bus, and the address is set by the core as (r1).

line 4: A Core move instruction, loading core register r2 from a memory location. Note that loading of R registers in parallel to an ISAP instruction is allowed (unlike D registers).

6.7.1.2 Working with Multiple ISAPs

In this case, each ISAP instruction should be specified by prefixing it with a unique label written before the brackets. This label overrides the default value set with the `ISAP_ID_default` directive.

The following syntax is used:

```
ISAP_ID1{ .. ISAP1 instructions .. }
ISAP_ID2{ .. ISAP2 instructions .. }
```

This method is preferred when multiple ISAP modules are used with the core, and are used interchangeably in the same code section.

In the following example, the two ISAPs that are connected in parallel are a Floating Point ISAP (FP) and an Image Processing ISAP (IP):

Example 6-5. Multiple ISAP coding

Two VLES lines that use an explicit ISAP ID string, for two different ISAPs:

```
mac d0,d1,d3 IP{isap_instruction k0,k1,k2}
mac d0,d1,d3 FP{isap_instruction k0,k1,k2}
```

In this example, three parallel instructions are used for each line:

1st - **mac** = a core instruction

2nd - **IP** or **FP** = instructs the ISAP controller to which ISAP the following instruction belongs

3rd - **isap_instruction** = a fictional ISAP instruction, used here for illustration purposes only

Note that the ISAP's registers names could have the same indication (k0,k1,k2), but are actually different registers because they belong to different ISAPs.

Multiple ISAPs in a Multi-Line VLES

In this example, there are multiple ISAPs connected to the core, therefore each ISAP has a unique ID string. The two ISAPs that are connected in parallel are a Floating Point ISAP (FP) and an Image Processing ISAP (IP):

```
[
mac d0,d1,d3
IP{isap_instruction k0,k1,k2}
]
[
mac d0,d1,d3
FP{isap_instruction k0,k1,k2}
]
```

In this example, in the first execution set the core performs a parallel MAC instruction and the Floating Point ISAP executes its own instruction. In the second execution set, the core performs a MAC instruction, and the Image Processing ISAP executes its own instruction.

6.7.2 An Example of the Definition Flexibility of an ISAP

When assembling a data move instruction to the ISAP, the assembler generates a parallel AGU instruction that will handle the address generation (for memory moves), immediate value generation or core register drive/sample. The ISAP portion of the instruction is responsible for ISAP register activation. These ISAP instructions are normally designated with MOVE mnemonics, as described in [Section 6.4, "ISAP Memory Access,"](#) on page 6-60.

However, the ISAP architect can define special move instructions that involve additional processing of the data before or after a memory accesses which is related to it. This example shows how flexible and powerful the ISAP can be, and demonstrates a MOVE instruction that includes other tasks in the same opcode:

```
nop {permute_lsb_set.21 (r0)+,k0} abs d0
```

This is similar example to that shown in [Section 6.7.1, “Identification of ISAP instructions,”](#) but instead of the ISAP `tsteq` instruction, a special move from a memory location to ISAP register `k0` is used. The special ISAP move instruction (called `permute_lsb_set .2l` in this case) can read 64-bit data, permute the bytes according to a certain definition, set the LSB, and write the result to register `k0`.

In this case, the ISAP assembler translates its move instruction to a core move instruction with dummy core registers `d0,d1` - `move .2l (r0)+,d0:d1`. This example is aimed to show how complex some specialized ISAP instructions may get, and why it is not desirable to constrain their syntax.

6.7.3 Conditional Execution

ISAP instructions can be conditionally executed by using the core `IFc` prefix instruction. By definition, the `IFc` mnemonics imply prefix predication. A VLES can have up to two `IFc` groups.

The following syntax conventions and limitations apply to conditionally executing ISAP instructions (See also Rule G.P.9):

- `IFc` mnemonic must be outside any ISAP clause (predication is the property of the core syntax, even if predicating only an ISAP clause as in the case of the `iff` in the above example)
- There can be two ISAP clauses per VLES, each belonging to a different `IFc` group.
- ISAP ALU instructions must all be in the same `IFc` group
- Implicit Core AGU instructions generated to support ISAP move instructions are subject to the same limitations as other core AGU instructions. This means for example:
 - If the VLES includes two `IFc` groups, and there are two ISAP move instructions, then the each ISAP move must be in a different `IFc` group.

The `IFc` predication convention is supported by ISAP interface signals that inform the ISAP of the predication status for each AGU instruction (associated with a data bus).

According to Rule G.P.9, it is not allowed to have two ISAP ALU instructions in two `IFc` groups. However, ISAP move instructions will cause the assembler to generate implicit core MOVE instruction, which could be in different `IFc` groups. For more details on how this works, see [Section 6.4, “ISAP Memory Access,”](#) on page 6-60.

Example 6-6. Conditional Execution Example

```
ift inc d0 {move.w (r0),k0} iff {tfra k0,k1 move.l (r1),k5}
```

The move instructions inside the ISAP brackets translates into an implicit core MOVE instructions, each conditioned by a different `IFc` condition.

Example 6-7. Conditional Execution Example

```
1 [ ift mac d0,d2,d4 mac d1,d3,d5
2   iff {alu_instruction k0,k1,k2
3     move_special.w k2,(r1)+}
4   move.l (r0)+,r2]
```



line 1: The ift (if true) prefix instruction indicates that the core MAC instructions will be executed only if the T bit is set.

lines 2,3,4: The iff (if false) prefix instruction indicates that the ISAP instructions on lines 2,3 (including the implicit MOVE generated for the move_special ISAP instruction) **and** the core MOVE.L instruction on line 4 will be executed if the T bit is cleared.

6.8 Programming Rules

This section overviews all SC140 programming rules that relate to ISAP instructions. These programming rules also appear in [Chapter 7, “Programming Rules,”](#) but are briefly summarized here as well.

ISAP-specific programming rules depend on the specific ISAP application should be described in the specific ISAP specification document, and are out of the scope of this document.

6.8.1 ISAP Functions that Interact With the Core

The functionality of ISAP instructions is in principle free to be defined as the ISAP architect wishes. However, some ISAP functions use core resources or affect core state. These functions usually have limitations associated with them, as will be described in the following sections:

- Read data from the data memory to ISAP register(s). An implicit SC140 AGU MOVE instruction is needed for this function.
- Write data from to ISAP register(s) to the data memory. An implicit SC140 AGU MOVE instruction is needed for this function.
- Exchange data between core registers and ISAP register(s). An implicit SC140 AGU MOVE instruction is needed for this function.
- Accept immediate data from the core into ISAP register(s). An implicit SC140 AGU MOVE instruction is needed for this function.
- Change the value of the SC140 core T bit in SR
- ISAP instructions could be predicated with the IFT/IFF prefix instructions like other core instructions.

In addition, there are other guidelines, as follows:

- ISAP instructions must have the same semantics of parallel execution like core instructions (for example destination operands do not affect source operands in the same VLES, etc.). In general, the guidelines as written in [Sections 7.1 VLES Sequencing Semantics](#) through [7.3 SC140 Pipeline Exposure](#) must be adhered to.
- The ISAP’s pipeline should be coupled to the SC140 core, which means that instructions that perform data memory accesses or data processing should have similar pipeline behavior like respective SC140 core AGU or DALU instructions. For a description on the SC140 pipeline, see [Chapter 5.1, “Pipeline.”](#)

6.8.2 Grouping rules for explicit ISAP instructions

G.G.2: up to 8 instruction words per VLES

G.G.3: One ISAP encoding word per VLES

G.G.4: A destination operand can only be updated by one source per VLES. In the context of the ISAP, it means the following limitations:

- The same core register being updated by the ISAP and another core or ISAP instruction
- A core address register updated by an implicit AGU instruction generated for supporting and ISAP instruction, and the same address register being updated by another instruction.
- The T bit being updated by the ISAP and another core or ISAP instruction

G.P.8: It is not allowed to group AGU instructions that use or update a data register (D0-D15) in the same VLES with an ISAP instruction.

G.P.9: ISAP ALU instructions must belong to the same IFc condition group.

6.8.3 Rules for implicit AGU instructions

As mentioned in [Section 6.4, “ISAP Memory Access,”](#) on page 6-60, when a data transfer instruction between the ISAP and the memory or core register is written inside the ISAP brackets, the core assembler creates an implicit MOVE instruction that will send the required address to the data memory.

The ramification of this is that when there is a MOVE-like instruction inside the ISAP brackets, all the rules that apply for “original core” MOVE-like instructions apply for the implicit MOVE-like instructions as well.

An example from Rule A.2, for an ISAP instruction, is shown below:

Example 6-8. MOVE rules with an implicit MOVE instruction from ISAP

The original rule requires (among other cases) a cycle difference between a MOVE-like instruction to an R register and it’s use as an AAU operand (see [Rule A.2](#) on page 7-17):

```
core_ins {move.l k0,r0}           ;core instruction, with ISAP move instruction
adda r0,r1                       ;not allowed
```

This is so because the core assembler generates the implicit MOVE instruction. In the example above, the effective assembler code for the first line will look like:

```
core_ins move.l d0,r0 {move.l k0,bus}
jmp      r0                       ;not allowed
```

Where the core ignores its own d0 part of the implicit move instruction. As can be seen, Rule A.2 prohibits this.

In a manner similar to the one shown, the following rules are relevant also to implicit AGU instructions:

G.G.5, G.P.1, G.P.4, G.P.5, G.P.6, A.1, A.2, A.4, T.1, SR.2, A.2a, A.5, A.6, D.7, A.1a.



6.8.4 Sequencing rules for T bit update

The ISAP has the ability to change the T bit as a destination of its instructions. The ISAP is less tightly coupled with the core, hence there the required dependency distance between The update of the T bit by the ISAP and usage by conditional core instructions is larger for non-DALU instructions.

T.2a: One VLES required between an ISAP instruction that updates the T bit and a conditional COF instruction

T.2b: 2 VLES required between an ISAP instruction that updates the T bit and a MOVET/F instruction

T.2c: 2 VLES required between an ISAP instruction that updates the T bit and an AGU instruction conditioned by IFT/F.

In addition, ISAP instructions that change the T bit or depend on it (via IFc) are subject to the same rules as are other core instructions:

T.1, D.2, D.3.



Chapter 7

Programming Rules

The SC140 has programming rules for correct construction and execution of assembly language code. These rules define the ability to group or sequence instructions that activate various execution units, because of their use of shared resources. This chapter describes the SC140 programming rules and guidelines for correct code construction. These programming rules must be followed to ensure that software applications produce expected results, and are compatible with future processor implementations. The SC140 assembler and simulator assist the programmer in conforming to these programming rules.

7.1 VLES Sequencing Semantics

The SC140 VLES execute in sequence according to the following semantics¹:

- One VLES finishes execution before the next VLES begins execution, with one exception being delayed change-of-flow instructions finish execution after the delay slot. This means:
 - The results of one VLES are immediately available to the next sequential VLES.
 - The latency of various SC140 instructions is not exposed in the assembly source code.
 - The number of cycles for a VLES is the number of cycles taken by its longest instruction.
- Even though the SC140 pipeline overlaps the execution of several VLES, the assembly source order for sequential VLES execution is enforced.

7.2 VLES Grouping Semantics

The SC140 instructions grouped in a VLES are executed according to the following semantics:

- Instructions to be executed together are explicitly grouped in a VLES by the programmer. This means:
 - The SC140 does not make any grouping decisions. However, it will take an illegal execution set exception if it decodes a grouping error.
 - Grouped instructions are placed on the same assembly source line, or on one or more lines surrounded by brackets [...].

1. Semantics is the meaning of language. In this context, the function of instructions, the relationship between instructions in a VLES, and the relationship between VLES in a program.

- All instructions in a VLES execute in parallel. This means:
 - The assembly source order of instructions and labels within an unconditional VLES does not change the results, with one exception being the assembly source order determines which instruction (if any) updates the carry bit. Because source order within an unconditional VLES is not required for correct code execution, the assembler sometimes reorders instructions in the VLES during the encoding process.
 - All instruction source registers and memory locations are read in parallel when the VLES starts execution. That is, the order that multiple reads occur does not change the results.
 - After all sources are read, all grouped instructions operate on them in parallel.
 - All instruction results are written to registers and memory locations in parallel when the VLES finishes execution. Parallel memory writes must write to different locations so the order that multiple writes occur does not change the results.
- Instructions in a VLES can execute conditionally. This means:
 - A VLES without conditional instructions executes all instructions unconditionally.
 - A VLES can have conditional instructions (such as Bc, Jc, MOVEc, TFRc, where c is the execution condition). The conditional instruction executes if its condition is true. If its condition is false, the instruction does not execute (becomes a NOP).
 - The IFc instruction conditionally executes all instructions that follow it (in assembly source order) until the next IFc instruction or the end of the VLES. Multiple IFc instructions in a VLES form subgroups of instructions that are conditionally executed based on their associated IFc condition. Multiple IFc subgroups having the same condition are in the same subgroup.
 - Each unconditional instruction in an IFc group or subgroup executes if its IFc condition is true. If its IFc condition is false, the instruction does not execute (becomes a NOP).
 - Each conditional instruction in an IFc group or subgroup executes if the instruction's condition and its IFc condition are both true. If either condition is false, the instruction does not execute (becomes a NOP).
- Only one carry-affecting instruction in a VLES may update the carry bit. This means:
 - A VLES can have multiple carry-affecting instructions that execute in parallel.
 - The carry-affecting instructions can execute unconditionally, or conditionally in an IFc group or subgroup.
 - The last (in assembly source order) carry-affecting instruction whose execution condition is true updates the carry bit, while the other instructions in the VLES do not affect the carry bit.
 - If no carry-affecting instructions execute, the carry bit is not affected.
- Even though the SC140 may execute instructions grouped in a VLES during different pipeline stages, the assembly source grouping for parallel instruction execution is enforced.

7.3 SC140 Pipeline Exposure

The SC140 has no hardware interlocks, so the pipeline is fully exposed during VLES execution. This is in direct contrast with the VLES sequencing and grouping semantics presented above, and is the motivation for the SC140 programming rules that follow. The programming rules hide the short pipeline latencies so they are not exposed in the assembly source code, saving hardware complexity and making SC140 code portable across different pipelines.

The SC140 software development tools (assembler and simulator) replace hardware interlocks with software detection of equivalent programming rules. This allows the programmer to write software that conforms to the VLES sequencing and grouping semantics defined above. The compiler code generation conforms to the programming rules by definition, since it always produces correct SC140 code.

7.4 Programming Rule Notation

Programming rules in this chapter use the following notation:

7.4.1 Grouping Rules

Grouping rules enforce the VLES grouping semantics within the **same** VLES by specifying that instruction “A” and instruction “B” cannot be “grouped in a VLES”. That is, “A” and “B” cannot execute in parallel.

Some grouping rules specify that instruction “A” cannot be in a VLES at a specific location. They use the notation that “A” cannot be “in a delay slot,” “in a short loop,” “at LA of the same long loop n,” or “in the first VLES of an exception service routine,” and so on.

7.4.1.1 Prefix Instructions

Prefix instructions are a unique instruction type, because they are encoded in a VLES prefix and are not dispatched to execution units. Any unique rule application is specifically noted in the rule definition.

7.4.1.2 Conditional Subgroups

Unless stated otherwise, the grouping rules in this chapter apply to the VLES as a whole regardless of any conditional instructions, IFC groups, or subgroups. Some rules apply independently to each IFC subgroup, and are specifically noted in the rule definitions. In this manual, the term “mutually exclusive subgroups” refers only to VLES having the IFT/IFF subgroups. IFT/IFA and IFF/IFA are not mutually exclusive subgroups.

7.4.1.3 Assembler Reordering

The assembler reorders instructions within a VLES for encoding efficiency. If more than one instruction in the VLES affects the carry bit C in SR, the assembler must keep the last (in assembly source order) carry-affecting instruction in each group or subgroup as last (highest word position) in its respective VLES encoding. This reordering conforms to the carry-update semantics described in [Section 7.2, “VLES Grouping Semantics,”](#) and is described in [Section 5.2.5, “Instruction Reordering Within an Execution Set.”](#)

7.4.2 Sequencing Rules

Sequencing rules enforce the VLES sequencing semantics by specifying the minimum distance between **different** VLES having instructions “A” and “B”. They use the notation “at least n VLES are required between” or “the minimum number of VLES between” instructions “A” and “B”. For a minimum VLES distance between “A” and “B”, the VLES having “A” and “B” are not included in this count.

7.4.2.1 Cycle Counts

Rules A.1 and A.2 specify a minimum cycle distance between “A” and “B” events. They use the notation “at least n cycles are required between when A and when B” occur. This takes into account the exact pipeline stages when “A” and “B” occur. By grouping a multiple cycle instruction with “A”, the minimum cycle distance may be met without scheduling additional VLES between “A” and “B”. For a minimum cycle distance between “A” and “B”, the cycles when “A” and “B” occur are not included in this count.

7.4.2.2 Conditional Execution

If instructions “A” or “B” are conditional or conditioned by an IFc instruction, they will execute only if their execution condition is true. If “A” and “B” have mutually exclusive execution conditions, their execution cannot violate a sequencing rule. The simulator knows if “A” and “B” execute from its simulation trace, and detects programming rules considering their conditional execution. However, the assembler cannot know the T bit state when “A” or “B” execute. So the assembler detects sequencing rules independent of conditional execution. That is, it assumes all conditional instructions always execute.

7.4.2.3 Simulator Execution Counts

A conditional VLES that is not executed (becomes a NOP) is counted by the simulator as one VLES for VLES-based sequencing rules. For cycle-based sequencing rules, a conditional VLES that is not executed (becomes a NOP) is counted by the simulator as one cycle.

7.4.3 Register Read/Write

In this chapter the programming rules use the notation “read” and “write” to refer to whole register sources and destinations, respectively. This notation applies to instruction source and destination operands, both explicit and implicit (implied), as specified in each instruction definition in [Appendix A.2, “Instructions,”](#) on page A-19.

7.4.3.1 Register Names

Some rules apply to selected registers — address, data, or program control. The rules specify the registers using the names given in [Table A-3: Register Abbreviations](#) on page A-3.

7.4.3.2 B Register Aliasing

The B0-7 base registers are the same registers as the R8-15 address registers in the AGU. For example, B0 and R8 have different source syntax and instruction encoding, but they are aliases to the same physical register. The rules always specify the Rn registers. The assembler and simulator detect the programming rules using either alias.

Example 7-1. B Register Aliasing

```
move.l d0,r8 move.l d1,b0 ;not allowed by G.G.4 - r8 and b0 are the same reg.
```

7.4.4 Status Bit Updates

In this chapter the programming rules use the notation “affected by” or “affect(s)” to refer to individual status bit sources and destinations, respectively. This notation applies to individual status bits as explicitly stated in each instruction definition in [Appendix A.2, “Instructions.”](#) The rules treat status bits as 1-bit registers.

7.4.5 Instruction Words

SC140 instructions can be one, two or three words long. All SC140 instructions have a base size of one word (16-bits). The second and third words of an instruction are called “extension words”.

7.4.6 MOVE-like Instructions

Instructions that access data during the Execute pipeline stage, including:

- All explicit MOVE instructions listed in [Table A-10: AGU Move Instructions](#) on page A-16
- VSL instructions listed in [Table A-10: AGU Move Instructions](#) on page A-16
- Pop/push instructions listed in [Table A-11: AGU Stack Support Instructions](#) on page A-16
- Bit mask instructions listed in [Table A-12: AGU Bit-Mask Instructions \(BMU\)](#) on page A-17

COF instructions that have implicit push/pop operations are not considered MOVE-like instructions.

In this chapter, a subset of this list is relevant if the rule applies to register sources only (POP is not relevant) or register destinations only (PUSH and BMTSTx are not relevant).

7.4.6.1 Address/Data Operands

In this chapter, MOVE-like instructions use pipeline-specific operand notation. “Data” operands refer to actual data being read from a source and written to a destination during the Execute pipeline stage. “Address” operands are read during the Address Generation pipeline stage to determine the address of a data operand in memory, and written during the Address Generation pipeline stage if an address register update is specified by the MOVE-like instruction. Address operands are not written for the (Rn) No Update and address pre-calculation addressing modes. When a programming rule applies to only one operand type, it will be stated in the rule definition. If not stated, the rule applies to both address and data operands of the MOVE-like instruction.

7.4.7 AGU Arithmetic Instructions

“AGU arithmetic instructions” are those instructions that execute during the Address Generation pipeline stage. This includes all instructions in [Table A-9: AGU Arithmetic Instructions](#) on page A-15. All SC140 instructions in this category end with the letter A (such as CMPEQA, ADDA, TFRA) with the following exceptions: IFA (a prefix instruction) and BRA (a non-loop COF instruction) are excluded.

7.4.8 Change-Of-Flow Destinations

A “change-of-flow (COF) destination” is any non-sequential change in the program counter (PC) register. COF destinations can be caused by COF instructions, hardware loop LA to SA iterations, and exceptions. Exceptions include instruction exceptions and hardware interrupts as described in [Section 5.8, “Exception Processing.”](#)

7.4.8.1 COF Instructions

A “COF instruction” specifies a (usually non-sequential) destination address that may implicitly write the program counter (PC) register. The COF result may be conditional on the T bit. All references to “COF instructions” in this manual include both the non-loop COF instructions listed in [Table A-13: AGU Non-Loop Change-of-Flow Instructions](#) on page A-17 and the loop COF instructions (BREAK, CONT, CONTD, and SKIPLS) from [Table A-14: AGU Loop Control \(Including Loop COF\) Instructions](#) on page A-18.

7.4.9 Delayed COF Instructions

A “delayed COF instruction” is a COF instruction that also executes the next sequential VLES. All COF instructions in this category end with the letter D.

Example 7-2. Delayed COF Instructions

```

BFD
BRAD
BSRD
BTD
CONTD
JFD
JMPD
JSRD
JTD
RTED
      RTSD
RTSTKD
  
```

7.4.9.1 Delay Slot

A “delay slot” is the next sequential VLES after a VLES having a “delayed COF instruction.”

7.4.10 Hardware Loops

The loop count “LCn” and start address “SAn” registers are described in [Section 5.4.1, “Loop Programming Model,”](#) on page 5-25.

The VLES addresses “LA, LA-1, LA-2” (relative to the loop end (last) address) and “SA, SA-1, SA-2” (relative to the loop start address) are defined in [Section 5.4.2, “Loop Notation and Encoding,”](#) on page 5-26. The minus “-” notation adjusts the VLES address earlier in the source code order.

The term “active loop” is defined in [Section 5.4.4, “Loop Nesting,”](#) on page 5-28.

All references to “in a loop” in this manual refer to both long and short loops, unless explicitly stated.

7.4.10.1 Enabled Loop

In a nested loop structure, more than one loop can be enabled at the same time. A loop n is enabled when its LFn bit in SR is set, where n is the loop index. The assembler’s static rule detection assumes loop n is enabled from the VLES after the VLES having the DOENn/DOENSHn instruction to LA of loop n (as specified by the LOOPENDn directive).

7.4.10.2 Enveloping Loop

In a nested loop structure, a loop “B” is nested inside another loop “A”. For this pair of loops, loop “B” is called a “nested loop” and loop “A” is called an “enveloping loop”. That is, loop “A” envelopes (surrounds or contains) loop “B”. This definition is relative to each loop pair. If another loop “C” is nested inside loop “B”, loop “C” becomes the nested loop and loop “B” becomes the enveloping loop for this loop pair.

Loops “A” and “B” are required to have ordered (but not adjacent) loop indexes. The assembler’s static rule detection assumes the boundaries of loop n are from the VLES after the VLES having the DOENn/DOENSHn instruction to LA of loop n (as specified by the LOOPENDn directive).

7.5 Static Programming Rules

Static programming rules are detected by examining the mnemonic/symbolic source code within the visibility of the assembler. Generally the assembler has visibility over a single source file with its INCLUDE files and MACRO expansions. The assembler detects all static rules in sequential VLES within its visibility.

7.5.1 Hardware Loop Detection

The assembler detects static rules for hardware loop iterations (LA to SA sequences) when all loop source code including the DOENn/DOENSHn instruction, LOOPSTARTn and LOOPENDn assembler directives for all nesting levels are within its visibility. The assembler rule detection makes the following assumptions:

- The loop flags (LFn and SLF) in SR are not changed within an active loop.
- Each loop n has one SA and one LA specified by the LOOPSTARTn and LOOPENDn directives, respectively.
- Loops enter at SA and exit from LA, except for exits using loop control instructions: BREAK and CONT/CONTD.

- The SAn register contains the starting address of the first VLES of long loop n.

These assumptions ensure the LOOPSTARTn and LOOPENDn directives are consistent with the hardware loop state machine’s decoding of the LPMARKx bits. The assembler may not detect hardware loop rules if these assumptions are violated.

7.5.2 General Grouping Rules

Rule G.G.1

Up to 6 instructions can be grouped in a VLES. Prefix instructions (IFc, LPMARKx, and NOP) are not counted for this rule.

Rule G.G.2

Up to eight instruction words (including prefix words) can be grouped in a VLES.

Example 7-3. VLES Word Count Exceeds Eight

```
jmpd r5  adr d0,d12  adr d1,d2  adr d2,d3  adr d5,d6  move.l #$12345678,r0
```

This example is not allowed. There are six instruction words plus two MOVE extension words and two prefix words.

Rule G.G.3

Instructions grouped in a VLES cannot exceed the available execution units. The maximum number of instructions in a VLES is:

- DALU instructions listed in [Table A-7: DALU Arithmetic Instructions \(MAC\)](#) on page A-13 and [Table A-8: DALU Logical Instructions \(BFU\)](#) on page A-14.
- Two AGU instructions, including:
 - All AGU arithmetic instructions listed in [Table A-9: AGU Arithmetic Instructions](#) on page A-15.
 - All AGU move instructions listed in [Table A-10: AGU Move Instructions](#) on page A-16.
 - All AGU stack support instructions listed in [Table A-11: AGU Stack Support Instructions](#) on page A-16
 - All control instructions listed in [Table A-13: AGU Non-Loop Change-of-Flow Instructions](#) on page A-17, [Table A-14: AGU Loop Control \(Including Loop COF\) Instructions](#) on page A-18, and [Table A-15: AGU Program Control Instructions](#) on page A-18.
- One bit-mask (BMU) instruction listed in [Table A-12: AGU Bit-Mask Instructions \(BMU\)](#) on page A-17. This instruction is also counted as an AGU instruction.
- One ISAP opcode (2-w prefix encoding) per VLES. An ISAP can be defined to support more than one instruction per ISAP opcode, [Section 6.3, “ISAP instructions and instruction encoding.”](#)
- This rule does not apply to prefix instructions listed in [Table A-16: Prefix Instructions](#) on page A-18 because they do not allocate execution units.

Example 7-4 Too Many AGU Instructions

```
bmtsts #$eb22,d5.h  move.w r2,(r0)+  move.w r2,(r5)  ;not allowed
```

Rule G.G.4

Instructions grouped in a VLES cannot write to the same register or affect the same status bit.

For mutually exclusive IFc subgroups in a VLES, this rule applies independently to each subgroup unless explicitly stated.

The less obvious cases are:

- Multiple COF instructions that implicitly write the PC register cannot be grouped in a VLES. This case applies to the whole VLES, independent of the T bit state.

Example 7-5 Duplicate PC Destinations

jmp _lbl3	bsr _lbl6	;not allowed
bt _label1	bf _label2	;not allowed
btd _label1	bfd _label2	;not allowed
jt r1	jf r2	;not allowed
jtd r1	jfd r2	;not allowed
ift bra _label1	iff bra _label2	;not allowed
ift brad _label1	iff brad _label2	;not allowed
ift jmp r1	iff jmp r2	;not allowed
ift jmpd r1	iff jmpd r2	;not allowed
ift jmp r0	iff rts	;not allowed
ift bra _label1	iff rts	;not allowed
ift bra _label1	iff break	;not allowed
ift jmp r1	ifa jf r2	;not allowed
ift bra _label1	iff brad _label2	;not allowed
ift jmpd r1	iff jmp r2	;not allowed

- Multiple writes of the same address pointer register Rn cannot be grouped in a VLES. The no update addressing mode (Rn) is not considered an address register write.

Example 7-6 Duplicate Address Pointer Register Destinations

move.w (r0)+,d0	move.w d1,(r0)+	;not allowed
move.w (r0)+,r0		;not allowed
move.l d0,r0	move.l (r0)+,d1	;not allowed
move.l d0,r8	move.l d1,b0	;not allowed - B register alias
pop r0	move.l (r0)+,d0	;not allowed
move.w (r0+\$6),r0		;allowed - no update mode
move.w (r0+n0),r0		;allowed - no update mode
move.w (r0),r0		;allowed - no update mode
move.w (r0),d0	move.w (r0)+,d1	;allowed - no update mode

- Multiple writes of the ESP or NSP stack pointer registers (implicitly using SP and OSP) cannot be grouped in a VLES. This rule applies independent of the EXP status bit.

Example 7-7 Duplicate Stack Pointer Destinations

pop d2	rts	;not allowed
pushn d0	tfra r1,osp	;not allowed
pop d1	tfra r0,sp	;not allowed
tfra r0,sp	tfra r1,osp	;allowed - writes different regs.

- Multiple instructions that write different portions of the same register cannot be grouped in a VLES.

Example 7-8 Duplicate Register Destinations

```

move.w #$1234,d0.h  move.w #$5678,d0.l      ;not allowed
bmset #3,sr.h      bmset #4,sr.l          ;not allowed
  
```

Note that `BMSET #3,SR.H` reads and writes the 32-bit SR register.

- MOVE-like instructions that write the SR or EMR register cannot be grouped in a VLES with instructions that affect individual status bits in the same register.

Example 7-9 Duplicate SR/EMR Register Destinations

```

pop sr      add d0,d1,d2      ;not allowed - C bit is written twice
ift pop sr  iff add d0,d1,d2  ;allowed
pop sr      bmtstc #$3,d0.l   ;not allowed - T bit is written twice
add d0,d1,d2 bmset #$0040,emr.l ;not allowed - DOVF is written twice
add d0,d1,d2 bmset #$0040,emr.h ;not allowed - DOVF is written twice
  
```

Note that `BMSET #3,SR.H` reads and writes the 32-bit SR register, while `BMTSTC #3,D0.L` affects only the T status bit in SR.

- Multiple instructions that affect the same status bit (T, VF0-3, DI, LF0-3, or SLF bits in SR) cannot be grouped in a VLES.

Example 7-10 Duplicate Status Bit Destinations

```

cmpeq d0,d1  tstgea.l r0      ;not allowed - multiple T bit updates
cmpeq d0,d1  bmtstc #$3,d0.l  ;not allowed - multiple T bit updates
  
```

Note that `BMSET #3,SR.H` reads and writes the 32-bit SR register, while `BMTSTC #3,D0.L` affects only the T status bit in SR.

- Rule G.G.4 also applies to core vs. ISAP instructions in addition to ISAP vs. ISAP instructions. For the latter case, the ISAP assembler is responsible for detecting the violation.

Rule G.G.4 Exceptions

- Two push or pop instructions that both implicitly write the SP register can be grouped in a VLES if they access different, De (even) and Do (odd), register fields. For the following groupings, the SC140 ensures that the SP register is written correctly. This case applies to the whole VLES.

Example 7-11 Dual Stack Pointer Destination Exception

```

push  De field  push  Do field  ;allowed
pushn De field  pushn Do field  ;allowed
pop   De field  pop   Do field  ;allowed
popn  De field  popn  Do field  ;allowed

ift push De field  ifa push Do field  ;allowed
  
```

- Two mutually exclusive writes to the same register (except the PC register) can be grouped in a VLES.

Example 7-12 Mutually Exclusive Register Destination Exception

```

ift  add #1,d0           iff  add #2,d0           ;allowed
movet r0,r1             movef r2,r1             ;allowed
tfrt  d0,d1             tfrf  d5,d1             ;allowed
ift  add #1,d0           ifa  tfrf d1,d0           ;allowed
ift  adda #1,r0          iff  adda #2,r0          ;allowed
ift  move.w (r0)+,d0     iff  move.w (r0)-,d0     ;allowed
[ift  move.w (r0)+,d0
   iff  {move_special (r0)-,d0} ] ;allowed
ift  tfra r1,r0          iff  tfra r2,r0          ;allowed
ift  push d0             iff  push d1             ;allowed
ift  movet r1,r0        movef r2,r0          ;allowed
    
```

- Instructions that affect different status bits can be grouped in a VLES. These include the C, T, S, VF0-3, DI, LF0-3, and SLF status bits in SR, and the DOVF bit in EMR. The G.G.4 rule treats instructions that affect these status bits as writing 1-bit destinations.

Example 7-13 Mutually Exclusive Status Bit Destination Exception

```

cmpeq d0,d1           add d0,d1,d2           ;allowed - T and C bit updates
di                    moves.f d0,(r0)+       ;allowed - DI and S updates
doen0 #5              max2vit d4,d2          ;allowed - LF and VF updates
    
```

- Multiple instructions that affect the C or S status bits in SR or the DOVF status bit in EMR can be grouped in a VLES. S and DOVF are “sticky” status bits are set by the logical OR of all executed instructions in a VLES that affect them. C is updated by only one instruction in a VLES - the last (in the assembly source order) carry-affecting instruction that actually executes in the VLES. This case applies to the whole VLES.

Example 7-14 Multiple C, S and DOVF Status Bit Destination Exception

```

add d0,d1,d2          asrr #4,d0              ;allowed -last instruction (asrr)
                                                            ;affects the C status bit in SR

[
  [moves.4f d0:d1:d2:d3,(r0)+
  moves.2f d4:d5,(r1)+ ] ;allowed - d0-d5
                                                            ;data affects S bit
    
```

Rule G.G.5

A data register Dn can be used as a source operand up to four times in a VLES. This includes all implicit sources, such as the accumulator source of a MAC instruction.

Example 7-15. DALU Register Use Exceeds Four Times

```

mac d2,d2,d2          add d2,d2,d3 ;not allowed - d2 used 5 times as a source
    
```

7.5.3 Prefix Grouping Rules

The following rules only apply to prefix-grouped VLES.

Rule G.P.1

Up to two extension words can be grouped in a VLES. This means:

- A three-word instruction can only be grouped in a VLES with one-word instructions.

Example 7-16 VLES Extension Words Exceed Two

```
[
  bmsset #ab34,d3.1 ; 1st extension word
  move.l #f0d0,vba ; not allowed - 2nd & 3rd extension word
]
```

- A two-word instruction can only be grouped in a VLES with one other two-word instruction and/or one-word instructions.

Example 7-17 Two-Word Instructions Exceed Two

```
[
  move.f #1234,d0 ; 1st extension word
  extract #8,#16,d0,d2 ; 2nd extension word
  extractu #8,#24,d0,d3 ; not allowed - 3rd extension word
  zxt.b d0,d0
]
```

The second and third words of an instruction are called “extension words.” An extension word can occur in the following cases:

- Some immediate values
- Some absolute addresses or offsets
- Bit mask instructions listed in [Table A-16: Prefix Instructions](#) on page A-18
- INSERT or EXTRACT/U instructions
- Integer double precision instructions such as IMAC_{xx}, IMAC_{xxxx}, IMPY_{xx}, IMPY_{xxxx} and IMPY.W

Rule G.P.3

The following instructions in each line are mutually exclusive, and cannot be grouped in a VLES. For mutually exclusive IFc subgroups in a VLES, this rule applies independently to each subgroup.

- MARK
- DEBUG and DEBUGEV
- DI and EI
- BREAK, CONT, CONTD, DOENn, DOENSHn, and SKIPLS

STOP, WAIT, and any COF instruction cannot be grouped in a VLES. This applies also to a VLES having two mutually exclusive IFc subgroups.

Example 7-18. VLES Has Mutually Exclusive Instructions

stop	wait	;not allowed
mark	mark	;not allowed
stop	mark	;allowed
ift doen1 #5	iff doen2 #4	;allowed
ift debug	iff debugev	;allowed
ift stop	iff wait	;not allowed
ift wait	iff bra _label	;not allowed

Rule G.P.4

An RTE/D instruction cannot be:

- Grouped in a VLES with another AGU instruction.
- In a VLES having two IFc subgroups.

Example 7-19. RTE Uses Both AAU

rted	inca r3	;not allowed
ift clr d0	ifa rte	;not allowed - two IFc subgroups
ift rte	impy d0,d1,d2	;allowed - one IFc group

Rule G.P.5

The same Nn or Mn register can be used as a data source operand of a MOVE-like instruction only once in a VLES. For mutually exclusive IFc subgroups in a VLES, this rule applies independently to each subgroup.



Example 7-20. Data Source Use of Nn and Mn Registers

move.l n0,d0	move.l n0,d1	;not allowed
ift move.l n0,d0	iff move.l n0,d1	;allowed
move.l n0,d0	move.l n1,d1	;allowed
move.l n0,d0	suba n0,r0	;allowed
move.l n0,d0	move.l (r0)+n0,d1	;allowed
move.l n0,d0	move.l (r0+n0),d1	;allowed
move.l (r0)+n0,d0	move.l (r1)+n0,d1	;allowed
move.l (r0)+n0,d0	move.l (r1+n0),d1	;allowed
move.l (r0+n0),d0	move.l (r1+n0),d1	;allowed
move.l n0,d0	vsl.2f d1:d3,(r0)+n0	;allowed
adda n0,n0	move.l (r0)+n0,d1	;allowed
adda n0,n0	move.l (r0+n0),d1	;allowed

Rule G.P.6

In a VLES having two IFc subgroups, each subgroup can have up to one AGU instruction and two DALU instructions. Prefix instructions (IFc, LPMARKx, NOP, and ISAP instructions) are not counted for this rule. However, if the core assembler adds implicit AGU instructions to support ISAP memory accesses and register transfers, this rule **does** apply to these implicit AGU instructions. For more details on how this works, see [Section 6.4, “ISAP Memory Access,”](#) on page 6-60.

Note that the overall number of DALU instructions in the entire VLES is restricted by [Section , “Rule G.G.3,”](#) .

Example 7-21. IFc Having Two Subgroups

ift add d0,d2,d3	iff move.w d3,(r4)	move.w d4,(\$8)	;not allowed
ift move.w d3,(r4)	iff add d0,d2	clr d4	;allowed
ift move.l d2,(r1)	iff add d3,d4	{isap_ins}	;allowed

Rule G.P.7

Up to two IFc subgroups (different conditions) can be grouped in a VLES. An IFc group or subgroup must have at least one instruction. An IFA subgroup (if present) must be the last (in the assembly source order) instructions in a VLES.

Example 7-22. IFA Subgroup Must Be Last Instructions

ift add d0,d1,d2	iff add d3,d4,d5	
ift	iff inc d0	;not allowed
inc d0	ift add d0,d1,d2	;not allowed
ifa inc d0	ift add d0,d1,d2	;not allowed
ift add d0,d1,d2	ifa inc d0	;allowed

Rule G.P.8

It is not allowed to group AGU instructions that use or update a data register (D0-D15) in the same VLES with an ISAP instruction.

This rule relates to independent AGU instructions, not to instructions that are implicitly generated by the assembler from ISAP instructions to support ISAP memory accesses and register transfers. For more details on how this works, see [Section 6.4, “ISAP Memory Access,”](#) on page 6-60.

Example 7-23. Core AGU instructions on same VLES as ISAP instructions

```

move.l (r0)+,d0 {INC K0} ; not allowed {INC K0} is an ISAP instruction
add d0,d1,d2 {INC K0} ; allowed, "add" is not an AGU instruction
{MOVE.L D0,K0} ; allowed. The core assembler generates an
; implicit AGU move from d0

```

Rule G.P.9

All ISAP ALU instructions in a VLES must belong to the same IFc group. ISAP instructions that generate implicit AGU instructions are subject to Rule G.P.6. See more on conditionally executed ISAP instructions in [Section 6.7.3, “Conditional Execution.”](#)

Example 7-24. ISAP instructions in same IFc group

```

ift {isap_one}      iff {isap_two}      ;not allowed
ift move.w d3,(r4)  iff {move_special} {isap_two} ;allowed
ift move.l d2,(r1)  iff add d3,d4      {isap_ins} ;allowed

```

7.5.4 AGU Rules

Rule A.1

At least two cycles are required between when an instruction writes the MCTL register and when an AGU instruction reads the R0-R7 registers with an address register update or address pre-calculation, or as an operand affected by a MCTL modifier field. This rule does not apply to R8-R15, or to R0-R7 using the no update (Rn) addressing mode.

Example 7-25. MCTL Write to R0-R7 Use

move.l	#\$12345678,mctl	;change MCTL
move.w	(r0)+,d0	;use MCTL, not allowed
move.l	#\$12345678,mctl	;change MCTL
nop		
move.w	(r0)+,d0	;use MCTL, not allowed
move.l	#\$12345678,mctl	;change MCTL
nop		
nop		
move.w	(r0)+,d0	;use MCTL, allowed
move.l	d0,mctl	;change MCTL
adda	r0,r1	;use MCTL, not allowed
move.l	d0,mctl	;change MCTL
move.w	d1,(r0+n0)	;use MCTL, not allowed
bmclr	#0,mctl.l	;change MCTL
move.w	(r0)+,d0	;use MCTL, not allowed
bmclr	#0,mctl.l	;change MCTL
move.w	(r1)+,d0	;use MCTL, not allowed
bmclr	#0,mctl.l	;change MCTL
move.w	(r5)+,d0	;use MCTL, not allowed
move.l	d0,mctl	;change MCTL
adda	r8,r1	;use MCTL, not allowed
move.l	d0,mctl	;change MCTL
adda	r1,r8	;no modifier mode, allowed
move.l	d0,mctl	;change MCTL
adda	#\$1234,r8,r1	;use MCTL, allowed
move.l	d0,mctl	;change MCTL
adda	#\$1234,r1,r8	;no modifier mode, not allowed
move.l	d0,mctl	;change MCTL
move.w	(r0),d0	;no update mode, allowed
move.l	d0,mctl	;change MCTL
move.w	(r8)+,d0	;no modifier mode, allowed

Rule A.2

At least one cycle is required between a MOVE-like instruction writing to an address register (Rn or Nn) as a data operand and when the same register is used in the following manner with the following instructions:

- An address operand of a MOVE-like instruction.
- A source operand of an AGU arithmetic instruction.
- An operand holding a target of a COF instruction ¹.

This rule does not apply when a MOVE-like instruction writes Bn or Mn and used for modulo calculation on Rn, by either MOVE or an AGU arithmetic instruction is the instruction immediately following the MOVE-like instruction (even in the case that Bn is referred to as Rn+8)². The rule does apply to writing to Rn (for n = 8÷15), using Bn-8 representation³.

Example 7-26. Rn, Nn, Mn Write to AGU Use

```

move.l  d0,r0
jmp     r0           ;not allowed

move.l  d0,r0
move.w  (r0),d1     ;not allowed

move.l  d0,b0
move.w  (r8),d1     ;not allowed (b0 is alias of r8)

move.l  d0,b0
move.w  (r0)+,d1    ;allowed - (but may contradict Rule A.2a due to r0 post inc)

move.l  d0,r0
bmclr.w #4,(r0)    ;not allowed

move.l  d0,b0
move.w  (r0),d1     ;allowed - no Rn write

move.l  d0,b0
adda   r9,d0        ;not allowed

move.l  d0,b3
adda   r11,d2       ;not allowed - (b3 is alias of r11)

move.l  d0,m0
move.w  (r0),d1     ;allowed - no Rn write

move.l  d0,m1
move.l  d1,(r0)+    ;allowed (but may contradict Rule A.2a due to r0 post inc)

move.l  d0,n0
move.b  (r0+n0),d1  ;not allowed

```

Rule A.3

A VLES having a JT/JF or BT/BF or TRAP instruction must be followed by another VLES. The last VLES in a program section cannot contain a JT/JF or TRAP instruction.

1. i.e. JMP Rn, JMPD Rn, JSR Rn, JSRD Rn, JT Rn, JTD Rn, JF Rn and JFD Rn, DOENn Rn or DOENSHn Rn
2. Writing to Bn and Mn is covered by dynamic rule A.2a.
3. B and R aliasing is applicable for other rule exceptions as well. See [Chapter 7.4.3.2, "B Register Aliasing,"](#) for more details.

If the VLES having a JT/JF or TRAP instruction is at the end of a program section, the following VLES must be a NOP. It cannot be data tables or uninitialized memory.

Rule A.4

An AGU arithmetic instruction that writes a Rn or Nn register, or a MOVE-like instruction that writes a Rn or Nn register as an address operand, cannot be grouped in a VLES with a MOVE-like instruction that reads the same register as a data operand. For mutually exclusive IFc subgroups in a VLES, this rule applies independently to each subgroup.

Example 7-27. Rn or Nn Write to MOVE-like Use

adda #\$5,r0	move.w r0,(\$100)	;not allowed
move.w (r0)+,d0	bmtsts #\$1234,r0.l	;not allowed
tfra r1,r0	push r0	;not allowed
tfra r1,n0	move.l n0,d0	;not allowed
tfra osp,r0	move.l r0,d0	;not allowed
move.w r0,(r0)+		;not allowed
move.w d0,(r8)+	move.l r8,d1	;not allowed
move.w d0,(r8)+	move.l b0,d1	;not allowed - b0 alias
vsl.2w d1:d3,(r0)+n0	move.l r0,d0	;not allowed
move.w (r2),r0	move.w r0,(r1)	;allowed
move.w r0,(r0)		;allowed - no Rn write
adda #>28,r6,r0	move.l (r0),r2	;allowed
adda #>28,r6,r0	move.l r2,(r0)	;allowed
adda #>5,n0	move.l d0,(r0)+n0	;allowed
adda #>5,n0	move.l (r0)+n0,d0	;allowed
adda #>5,n0	move.l d0,(r0+n0)	;allowed
adda #>5,n0	move.l (r0+n0),d0	;allowed
ift adda #<1,r0	iff move.l r0,d0	;allowed
iff adda #<1,r0	ifa movet r0,r7	;allowed

A DOENn or DOENSHn, CONT, or CONTD instruction that writes an LCn register cannot be grouped in a VLES with a MOVE-like instruction that reads the same register. For mutually exclusive IFc subgroups in a VLES, this rule applies independently to each subgroup.

Example 7-28. LCn Write to MOVE-like Use

doen0 r0	move.l lc0,d0	;not allowed
doensh0 #5	push lc0	;not allowed
cont	move.l lc1,d3	;not allowed
contd	push lc2	;not allowed

Rule A.7

A RTSTK or RTSTKD instruction cannot be grouped in a VLES with a MOVE-like instruction that reads the EMR register. For mutually exclusive IFc subgroups in a VLES, this rule applies independently to each subgroup.

Example 7-29. NMID Update to EMR Read

```
rtstk          move.l emr,d0          ;not allowed
```

7.5.5 Delayed COF Rules

Rule D.1

The following instructions are not allowed in a delay slot:

- COF instructions
- STOP and WAIT
- DI
- DEBUG

Example 7-30. Instructions in a Delay Slot

```
jmpd r1  
jmp r2          ;not allowed
```

Rule D.2

Core or ISAP instructions that read or write the SR register, affect status bits in SR, or are affected by status bits in SR are not allowed in a RTED delay slot.

This rule applies to instructions that use the stack pointer SP (implicitly or explicitly) and other stack pointer OSP, since SR affects which stack pointer is used (EXP status bit).

Example 7-31. Instructions in a RTED Delay Slot

```

rted
move.l d0,sr                ;not allowed

rted
move.l sr,d0                ;not allowed

rted
bmset #1,sr.l              ;not allowed

rted
rol d0                      ;not allowed, affected by SR[C]

rted
push d0                     ;not allowed, affected by SR[EXP]

rted
bmclr.w #64,(sp-8)         ;not allowed, affected by SR[EXP]

rted
bmtsts.w #64,(r0)         ;not allowed, affects SR[T]

rted
tfra r0,osp                ;not allowed, affected by SR[EXP]

rted
tfra sp,r0                 ;not allowed, affected by SR[EXP]

rted
ift clr d0                 ;not allowed, affected by SR[T]

rted
tfra r0,r1                 ;allowed

rted
{tsteq k0}                 ;changes T bit based on ISAP register - not allowed

rted
bmclr.w #64,(r0)          ;allowed
    
```

Rule D.3

Core or ISAP instructions that write the SR register or affect status bits in SR cannot be grouped in a VLES with a RTE/RTED instruction.

Example 7-32. RTE/D with SR Updates

```

rte    add d0,d1,d2        ;not allowed - affects the carry bit in SR
rte    {tsteq k0}         ;not allowed - affects the T bit in SR
    
```

Rule D.4

Instructions that read the PC register (implicitly or explicitly) as a source operand are not allowed in a RTED/RTSD/RTSTKD delay slot. This rule does not apply to the MARK instruction that reads the PC register for the EOnCE trace buffer.

Example 7-33. PC Read in a Return Delay Slot

```

rted
adda pc,r0          ;not allowed

rtsd
dosetup0 _label    ;not allowed

```

Rule D.5

A MOVE-like instruction that writes the SR register cannot be grouped in a VLES with a BSR, BSRD, JSR or JSRD instruction. For mutually exclusive IFc subgroups in a VLES, this rule applies independently to each subgroup.

Example 7-34. SR Write with a Subroutine Call

```

pop sr          jsr r0          ;not allowed

```

Rule D.5a

A MOVE-like instruction that writes the SR register is not allowed in the delay slot of a BSRD or JSRD instruction.

Example 7-35. SR Write in BSRD or JSRD Delay Slot

```

bsrd _label
pop sr          ;not allowed

```

Rule D.6

Instructions that read or write the SP register are not allowed in the delay slot of delayed return (RTSD, RTED, , and RTSTKD) instructions. This rule also applies to implicit SP register writes (push and pop instructions).

Example 7-36. SP Use in Return Delay Slots

```

rtsd
tfra r0,sp        ;not allowed

rted
tfra sp,r0        ;not allowed

rtsd
tfra r0,osp       ;allowed

rtstk
tfra osp,r0       ;allowed

rtstk
pop d0            ;not allowed

```

Rule D.8

A MOVE-like instruction that reads the SR register is not allowed in the delay slot of a CONTD instruction.

Example 7-37. SR Read in a CONTD Delay Slot

```
contd _label
move.l sr,d0      ;not allowed
```

Rule D.9

Instructions that read the EMR register are not allowed in the delay slot of a RTED, or RTSTKD instruction.

Example 7-38. EMR Use in Return Delay Slots

```
rtstk
move.l emr,d0     ;not allowed

rted
bclr #$fffb,emr.l ;not allowed
```

7.5.6 Status Bit Rules

Rule T.1

At least one VLES is required between an instruction that affects the T status bit in SR and an AGU instruction in an IFT/IFF group or subgroup. This rule does not apply to AGU instructions in an IFA subgroup.

Example 7-39. T Bit Update to IFT/IFF AGU Use

```
tsteg d0
ift move.l r0,d1      ; not allowed

tsteg d0
nop
ift move.l r0,d1      ; allowed

tsteg d0
ift mac d0,d1,d2      ; allowed

tsteg d0
ift mac d0,d1,d2     ifa move.l r0,d1 ; allowed
```

Rule T.2.a

At least one VLES is required between an ISAP instruction that affects the T status bit in SR and a conditional COF instruction.

Example 7-40. T Bit Update by ISAP and COF

```

{tsteq k0}                ; tsteq is an ISAP instruction that
                           updates the T bit
jt r0                     ; not allowed

{tsteq k0}
nop
jf _destination          ; allowed

```

Rule T.2.b

At least two VLES are required between an ISAP instruction that affects the T status bit in SR and a MOVET/MOVEF instruction.

Example 7-41. T Bit Update by ISAP and MOVET/MOVEF

```

{tsteq k0}                ; tsteq is an ISAP instruction that
                           updates the T bit
movet r0,r1              ; not allowed

{tsteq k0}
nop
nop
movet r1,r2              ; allowed

```

Rule T.2.c

At least two VLES are required between an ISAP instruction that affects the T status bit in SR and an AGU instruction in an IFT/IFF group or subgroup. This rule does not apply to AGU instructions in an IFA subgroup.

Example 7-42. T Bit Update by ISAP and IFT/IFF

```

{tsteq k0}                ; tsteq is an ISAP instruction that
                           updates the T bit
ift move.l d0,d1         ; not allowed

{tsteq k0}
nop
nop
ift move.l d1,d2         ; allowed

```

Rule SR.2

At least two VLES are required between a MOVE-like instruction that writes the SR register and an instruction affected by a status bit in SR.



This rule applies to instructions that use the stack pointer (implicitly or explicitly), since SR affects which stack pointer is used (EXP status bit).

A MOVE-like instruction that writes the SR register may be followed by a MOVE-like instruction that reads the SR register, if not affected by a SR status bit.

The assembler-mapped instruction CLR Dn is never affected by SR status bits, even though it is implemented as SUB Da, Da, Dn. Therefore, this rule applies to the SUB instruction, but not to CLR (SUB Da, Da, Dn is taken as CLR in this context).

Example 7-43. SR Write to SR Status Bit Use

```

bmclr  #ffff,sr.h      ;change SR
move.w  #1234,d0       ;allowed, not affected by SR

bmclr  #ffff,sr.h      ;change SR
rol     d0             ;not allowed, affected by SR[C]

bmclr  #ffff,sr.h      ;change SR
nop
rol     d0             ;not allowed, affected by SR[C]

bmclr  #ffff,sr.h      ;change SR
nop
nop
rol     d0             ;allowed

bmclr  #<1,sr.l        ;change SR
nop
push   d0             ;not allowed, affected by SR[EXP]

bmclr  #<1,sr.h        ;change SR
nop
ift    clr d0         ;not allowed, affected by SR[T]

bmtstc #0001,sr.l      ;read SR, affects SR[T], not a SR write
add    d1,d5,d1       ;allowed

pop     sr
bmsset #a,sr.l         ;allowed

move.l  d0,sr
move.l  sr,d5         ;allowed

bmsset #a,sr.h
and.w  #1234,(sp-8)   ;not allowed

bmsset #a,sr.l
and.w  #1234,(sp-8)   ;not allowed

bmsset #a,sr.l
bmsset #b,sr.h        ;allowed

move.l  d1,sr
move.l  d0,(sp+4)     ;not allowed

move.l  d1,sr
move.l  d0,(r0+4)     ;allowed

move.l  d1,sr         ;change SR
nop
clr    d0             ;allowed, clear not effected by SR

move.l  d1,sr         ;change SR
nop
sub    d1,d2,d3       ;not allowed, sub effected by S0,S1

move.l  d1,sr         ;change SR
nop
sub    d1,d1,d3       ;allowed, this is a CLR
    
```

Rule SR.3

At least one VLES is required between a MOVE-like instruction that writes the SR register and the following instructions that affect status bits in SR:

- DI and EI
- DOEN_n and DOENSH_n
- CONT/D, BREAK, and SKIPLS

Example 7-44. SR Write to SR Status Bit Update

```

move.l #<1, sr
di                                     ; not allowed

bmclr  #$ffff, sr.h
doen0  #<10                               ; not allowed

pop    sr
cont   _next                               ; not allowed

```

Rule SR.4

At least two VLES are required between an instruction that affects the DOVF status bit in EMR and a MOVE-like instruction that reads or writes the EMR register.

The assembler-mapped instruction CLR D_n never affects the DOVF status bit, even though it is implemented as SUB Da, Da, D_n. Therefore, this rule applies to the SUB instruction, but not to CLR (SUB Da, Da, D_n is taken as CLR in this context).

Example 7-45. DOVF Update to SR Read or Write

```

bmset  #$4,emr.l
move.l  emr,d2           ;allowed

move.l  #$4,emr
move.l  emr,d0           ;allowed

move.l  #$4,emr
move.l  d0,emr           ;allowed

move.l  d1,emr
move.l  emr,d0           ;allowed

add  d0,d1,d2           ;overflow may set DOVF bit
nop
move.l  emr,d3           ;not allowed

adr     d3,d4           ;overflow may set DOVF bit
nop
move.l  d0,emr           ;not allowed

bmset  0,$01,emr.l      ;read and write EMR register
nop
move.l  d0,emr           ;allowed

sub  d0,d1,d2           ;overflow may set DOVF bit
nop
move.l  emr,d3           ;not allowed

clr  d0                 ;DOVF bit not affected
move.l  emr,d3           ;allowed

sub  d1,d1,d0           ;This is a CLR. The DOVF bit is not affected
move.l  emr,d3           ;allowed
    
```

Rule SR.4a

Instructions that affect the DOVF status bit in EMR can't be grouped with a MOVE-like instruction to SR or with an RTE-like instruction (RTE/D).

The only exception for this rule is for Bit-Mask instructions on SR for which it is ensured that the value of the OVE bit in SR is not changed.

Example 7-46. DOVF Update grouped with Move-like SR updates

```

bmclr  #$0010,sr.h      mac  d1,d2,d3           ; Not allowed
bmchg  #$0010,sr.h      mac  d1,d2,d3           ; Not allowed
bmclr  #$0001,sr.h      mac  d1,d2,d3           ; Allowed (OVE not changed)
bmchg  #$0010,sr.l      mac  d2,d2,d3           ; Allowed (OVE not changed)
move.l  #$100000,sr     mac  d0,d1,d2           ; Not allowed
move.l  d0,sr           mac  d0,d1,d2           ; Not allowed
pop  sr                 mac  d0,d1,d2           ; Not allowed
rted                    mac  d0,d1,d2           ; Not allowed
    
```

Rule SR.7

The following instructions that affect status bits in SR cannot be grouped in a VLES with a MOVE-like instruction that reads the SR register:

- BREAK
- CONT/CONTD
- DI and EI
- DOENn and DOENSHn
- SKIPLS

For mutually exclusive IFc subgroups in a VLES, this rule applies independently to each subgroup.

Example 7-47. Status Bit Update with SR Read

doen0 #5	move.l sr,d0	;not allowed
di	push sr	;not allowed
skipls _dest	bmtsts #4,sr.h	;not allowed

7.5.7 Loop Nesting Rules

Rule L.N.1

Nested loops cannot have the same LA.

Example 7-48. Nested Loops with the Same LA

...		
move.w r3,(r4)		;LA
loopend1		
loopend0		;not allowed

Rule L.N.2

A loop body *n* must be surrounded by the LOOPSTART_{*n*} and LOOPEND_{*n*} assembly directives, and can only be nested inside a loop body having a smaller index.

Example 7-49. Nested Loops with Ordered Index

```

doen1    #count1
move.w   #num,d1
move.l   #mem_1,r1
move.w   #offset,n0

loopstart1
label1
inc      d1
dosetup0 label2
doen0    #count2
move.w   #num,d2

loopstart0           ;not allowed
label2
inc      d2
impyuu   d1,d2,d3
move.w   d3,(r1)+
loopend0

nop
loopend1

```

Rule L.N.3

A DOEN_{*n*}/DOENSH_{*n*} instruction having a different loop index and any LOOPEND directive cannot come between the DOEN_{*n*}/DOENSH_{*n*} instruction and LOOPSTART_{*n*} directive of loop *n*.

Also, it is not allowed to place a DOENSH instruction with any index between the DOEN_{*n*} and its respective LOOPSTART_{*n*} directive, or a DOEN instruction with any index between the DOENSH_{*n*} instruction and its respective LOOPSTART_{*n*} directive.

Example 7-50. Nested DOEN_{*n*}/DOENSH_{*n*} Instructions

```

count2 equ 5
...
move.w   #count2,d6
dosetup0 label2
doen0    d6
doen1    #2           ; not allowed

loopstart0
doen1    #5
doen1    #6           ; allowed

loopstart1

```

Example 7-51. DOENn instruction following DOENSHn Instruction

```

doensh0 #3
doen0 #3 dosetup0 _loop_start    ; not allowed (SLF isn't reset)
nop
nop
_loop_start
loopstart0
move.l #0,d0
move.l #1,d0
move.l #2,d0                      ;instruction should be in the loop, but isn't!
loopend0

```

Example 7-52. LOOPEND between DOEN and LOOPEND

```

doen2 #3
dosetup2 L_1
nop
L_1
loopstart2
nop
nop
doen3 #3          ;not allowed: problem created here
dosetup3 L_2
nop
nop
nop
loopend2         ;problem becomes apparent here
nop
L_2
loopstart3
nop
nop
nop
loopend3

```

Example 7-53. Changing a loop type

```

doensh0 #3
doen0 #3          ; not allowed to change loop type
dosetup0 _loop_start
nop
nop
_loop_start
loopstart0
move.l #0,d0
move.l #1,d0
move.l #2,d0
loopend0

```

7.5.8 Loop LA Rules

Rule L.L.1

The following instructions are not allowed at LA-1 or LA of a long loop:

- COF instructions
- STOP and WAIT
- DI
- DEBUG

Example 7-54. Instructions at the End of Long Loops

```
    move.w  #count2,d6
    dosetup0 label2
    doen0   d6
    move.w  #1,d1
    move.w  #2,d2
    move.w  #3,d3
    move.w  #4,d4

    loopstart0
label2 inc d1
      inc d2
      inc d3
      inc d4
      wait           ;not allowed
    loopend0
```

Rule L.L.2

A DOENn or MOVE-like instruction that writes a LCn register is not allowed at LA-2, LA-1, or LA of the same long loop n.

Example 7-55. LCn Write at the End of Long Loop n

```
    doen1 #5           ;not allowed
    move.w  d3,(r1)+
    loopend1
```

Rule L.L.3

The following instructions are not allowed in a short loop:

- COF instructions
- STOP and WAIT
- DI
- DEBUG
- DOENn/DOENSHn
- MOVE-like instructions that read any LCn register
- MOVE-like instructions that write any LCn register
- MOVE-like instructions that read the SR register
- MOVE-like instructions that write the SR register

This rule does not apply to other instructions that affect status bits in SR.

Example 7-56. Instructions in Short Loops

```
doensh0 #$10
nop
loopstart0
jmp end           ;not allowed
loopend0
```

```
doensh1 #count2
move.w #num,d2
loopstart1
doen1 #5         ;not allowed
loopend1
```

Rule L.L.4

The LA of a short loop cannot be at LA-1 of a long loop.

Example 7-57. Short Loop LA at the End of a Long Loop

```
dosetup0 label1
doen0 #count1
move.w #num,d1
nop
nop
loopstart0
nop
nop
label1 inc d1
doensh1 #count2
move.w #num1,d2
loopstart1
label2 inc d2      impyuu d1,d2,d3           ;not allowed
loopend1
nop
loopend0
```

Rule L.L.5

A MOVE-like instruction that writes the SR register is not allowed at LA-4, LA-3, LA-2, LA-1, or LA of a long loop.

Rule L.L.6

A MOVE-like instruction that writes the SR register is not allowed at SA-2 or SA-1 of a short loop.

7.5.9 Loop Sequencing Rules**Rule L.D.1**

At least one VLES is required between the following instructions that write any LCn register and the SKIPLS instruction:

- Any DOENn Dn (data register)
- Any DOENSHn Dn (data register)
- MOVE-like instruction that writes any LCn register

Example 7-58. LCn Write to SKIPLS Instruction

```
doen0 d2
skipls label4           ; not allowed
```

Rule L.D.2

The minimum number of VLES between the following instructions that write a LCn register and LA of the same long loop n is:

- DOENn Rn or #x: three VLES (address register or immediate value)
- DOENn Dn: four VLES (data register)
- MOVE-like instruction that writes a LCn register: four VLES

Example 7-59. LCn Write at the End of Long Loop n

```
move.w #3,d8
dosetup1 label1
doen1 d8           ;not allowed
nop
loopstart1
label1 inc d3
inc d4
inc d5
loopend1
```

Rule L.D.3

The minimum number of VLES between the following instructions that write a LCn register and SA of the same short loop n is:

- DOENSHn Rn or #x: one VLES (address register or immediate value)
- DOENSHn Dn: two VLES (data register)
- MOVE-like instruction that writes a LCn register: two VLES

Example 7-60. LCn Write at the Start of Short Loop n

```

move.w #3,r0
doensh0 r0           ;allowed
move.l d1,lc0       ;not allowed
move.w #2,d2
loopstart0
inc d1
loopend0

```

Rule L.D.5

The minimum number of VLES between an instruction that writes any LCn register and a CONT/CONTD instruction is:

- Any DOENn Rn or #x : one VLES (address register or immediate value)
- Any DOENn Dn: two VLES (data register)
- MOVE-like instruction that writes any LCn register: two VLES

Example 7-61. LCn Write to CONT/D Instruction

```

doenl #5
...
loopstart1
loop1 nop
move.l d0,lc1
nop
cont label           ;not allowed
nop
nop
loopend1

```

Rule L.D.6

A MOVE-like instruction that writes a SAn register is not allowed at (LA-3), (LA-2), (LA-1), and LA of the same long loop n.

Example 7-62. SAn Write at the End of Long Loop n

```

loopstart0
...
doen1 #5
...
loopstart1
...
dosetup0 _addr           ;not allowed
loopend1
nop
loopend0
    
```

Rule L.D.7

At least one VLES is required between an instruction that writes any SAn register and a CONT/CONTD instruction.

Example 7-63. SAn Write to CONT/D Instruction

```

doen1 #5
dosetup1 label1
loopstart1
label1 cont label2           ;not allowed
inc d0
move.w #$23,d2
move.w #$beef,d3
loopend1
label2 inc d1
    
```

Rule L.D.8

A MOVE-like instruction that reads a LCn register is not allowed at the (LA-3), (LA-2), (LA-1), and LA of the same long loop.

Rule L.D.9

At least one VLES is required between a MOVE-like instruction that reads a LCn register and SA of the same short loop n.

Example 7-64. LCn Read at the Start of Short Loop n

```

doensh0 #$10
push lc0                   ;not allowed
loopstart0
inc d0
loopend0

doensh0 #$10
push lc1                   ;allowed
loopstart0
inc d0
loopend0
    
```

7.5.10 Loop COF Rules

Rule L.C.1

A COF instruction cannot have a COF destination that is LA-1 or LA of a long loop, or LA of a 2-VLES short loop. This rule does not apply to loop COF instructions (BREAK, CONT, CONTD and SKIPLS) in a nested loop having a COF destination that is LA-1 or LA of an enveloping loop.

Example 7-65. COF Destination to Loop Delay Slots

```

doenshl #5
...
cmpeq.w #3,d0
jf _dest ;not allowed
inc d0
loopstart1
inc d0
_dest add d1,d2,d3
loopend1
  
```

Rule L.C.2

COF instructions, WAIT and STOP are not allowed at LA-2 of a long loop.

Example 7-66. COF Instructions at LA-2 of a Long Loop

```

dosetup1 label1
doenl #n2
move.l #mem_l1,r1
move.l #mem_l2,r0
loopstart1
label1 inc d1
jsr r1 ;LA-2, not allowed
add d1,d2,d3 ;LA-1
move.w d3,(r0) ;LA
loopend1
bra label2
  
```

Rule L.C.3

A Bc or Jc instruction is not allowed at SA-1 of a short loop.

Example 7-67. Bc/Jc at SA-1 of a Short Loop

```

cmpgt d4,d3
nop
iff doensh3 #count2
bt _dest ;SA-1, not allowed
loopstart3
inc d2
loopend3
...
_dest inc d2
  
```

Rule L.C.5

A Bc or Jc instruction is not allowed at LA-3 of a long loop.

Example 7-68. Bc/Jc at LA-3 of a Long Loop

```
dosetup1 label7
move.w #0,d1
doen1 #5
loopstart1
label7 inc d1
      bf label6           ;LA-3, not allowed
      inc d2
      inc d3
      inc d4
      loopend1
```

Rule L.C.7

A loop COF instruction (BREAK, CONT, CONTD, or SKIPLS) in an enabled loop n cannot have a COF destination in the same loop n.

Example 7-69. Loop COF Destination in the Same Loop

```

        dosetup3 label1
        doen3 #5
        loopstart3
label11 inc d1
        inc d2
        break label2           ;not allowed
        inc d3
        inc d4
label12 inc d5
        loopend3

        dosetup3 label1
        doen3 d0
        nop
        skip1s label2         ;allowed
        loopstart3
label11 inc d1
        inc d2
        inc d3
        inc d4
        inc d5
        loopend3
label12 nop

        dosetup2 label1
        doen2 #6
        nop
        loopstart2
label11 cont next           ;not allowed
        nop
        inc d0
        dosetup3 label2
        doen3 #5
        loopstart3
label12 inc d1
next   inc d2
        inc d3
        inc d4
        inc d5
        loopend3
        nop
        loopend2
        nop
  
```

Rule L.C.9

A loop COF instruction (BREAK, CONT, CONTD, or SKIPLS) cannot have a COF destination that is one VLES before two consecutive VLES that are both LA of long loops.

Example 7-70. Loop COF at End of Nested Long Loops

```

doen0 #5
...
loopstart0
...
doen1 #10
...
loopstart1
...
doen2 d0
...
skipls _dest          ; not allowed
loopstart2
...
loopend2
...
_dest
nop
nop                   ; last address of long loop 1
loopend1
nop                   ; last address of long loop 0
loopend0

```

Rule L.C.10

A BSR, BSRD, JSR, or JSRD instruction cannot have a COF destination that is LA-2 of a long loop or SA of a short loop.

Example 7-71. Subroutine Call to End of Loops

```

dosetup0 label1
doen0 d1
nop
nop
loopstart0
label1 nop
nop
jsr label2           ;not allowed
nop
nop
inca r1
label2 inca r7       ;LA-2
add d1,d2,d3        ;LA-1
move.w d3,(r0)      ;LA
loopend0

```

Rule L.C.11

A delayed COF instruction is not allowed at LA-3 of a long loop.

Example 7-72. Delayed COF at LA-3 of a Long Loop

```

    jmpd _dest          ;not allowed
    nop
    nop
    nop
    loopend0

```

Rule L.C.12

A delayed COF instruction is not allowed at SA-1 of a short loop.

Example 7-73. Delayed COF at SA-1 of a Short Loop

```

    jmpd _dest          ;not allowed
    loopstart0
    nop
    loopend0

```

7.5.11 General Looping Rules

Rule L.G.3

A MOVE-like instruction that reads the SR register is not allowed at the (LA-3), (LA-2), (LA-1), and LA of any long loop.

Example 7-74. SR Read to LA of Any Long Loop

```

    dosetup1 label1
    doen1    #5
    loopstart1
label1
    inc d1
    move.l sr,d0          ;not allowed
    inc d2
    move.l #mem_l1,r1
    move.l #mem_l2,r0
    loopend1

```

Rule L.G.4

At least one VLES is required between a MOVE-like instruction that reads the SR register and SA of any short loop.

Example 7-75. SR Read to SA of Any Short Loop

```

    doensh0 #$10
    push sr              ;not allowed
    loopstart0
    inc d0
    loopend0

```



Rule L.G.5

A loop having one or two VLES must be enabled by a DOENSHn instruction. A loop having three or more VLES must be enabled by a DOENn instruction.

Example 7-76. Enabling Short and Long Loops

```

doenl #5           ; not allowed
dosetupl label1
skipls label2
loopstartl
label1
move.l d1,(r1)+
addnc.w #1,d1,d1
loopendl
label2 nop

```

7.6 Dynamic Programming Rules

Dynamic programming rules cannot be detected by the assembler examining the source code, because they depend on the run-time execution trace. These rules can be detected by the simulator examining the SC140 visible and hidden registers and simulation trace. The simulator detects all dynamic rules that occur during its execution. The simulator cannot detect programming rules in binary encodings that were not executed.

Dynamic rule detection depends on the test coverage of the programmer's test suite. Programmers should exercise all COF destinations, exception service routines, and system configurations (such as conditional assembly directives) in the simulation trace so the simulator detects all dynamic programming rules. The simulation trace should also exercise all data dependencies of conditional COF instructions (such as Bc and Jc), conditional instructions (MOVEc and TFRc) and conditional groups or subgroups (IFc).

7.6.1 AGU Dynamic Rules

Rule A.2a

At least one cycle is required between when a MOVE-like instruction writes Bn or Mn register as a data operand and a MOVE or an AGU arithmetic instruction using the same register for a modulo calculation (even in the case that Bn is referred to as R_{n+8}). This rule applies only if these registers are actually used in a modulo calculation as determined by the actual value in MCTL.

Example 7-77. Bn, Mn Write to AGU Use

```

move.l #$000000f, MCTL           ; r0 only has modulo addressing
move.l d0,b0
move.w (r0)+,d1                 ; Not allowed
pop d0,m1
adda r9,r1                       ; Allowed

```

7.6.2 Memory Access Rules

Rule A.5

Only one memory write instruction to the same location can be grouped in a VLES. Multiple memory write instructions grouped in a VLES must write to different locations, so the order that multiple writes occur does not change the memory results. If this is not done, the memory contents of the accessed locations are undefined. For mutually exclusive IFc subgroups in a VLES, this rule applies independently to each subgroup.

Example 7-78. Multiple Memory Writes to the Same Location

<code>move.l d0,(r0)</code>	<code>move.l d1,(r0)</code>	<code>;not allowed</code>
<code>move.l d0,(r0+n0)</code>	<code>move.l d1,(r0+n0)</code>	<code>;not allowed</code>
<code>move.l d0,\$100</code>	<code>move.b d1,\$103</code>	<code>;not allowed</code>
<code>move.l d0,(r0)</code>	<code>move.l d1,(r1)</code>	<code>;not allowed if addresses overlap</code>
<code>move.l d0,(r0)</code>	<code>move.b d1,(r0+n0)</code>	<code>;not allowed if addresses overlap</code>
<code>move.l d0,(r0)</code>	<code>move.w d1,(r1+\$6)</code>	<code>;not allowed if addresses overlap</code>

Rule A.6

A memory read instruction having a pre-calculated address cannot be grouped in a VLES with a memory write instruction not having a pre-calculated address to the same memory location. These memory accesses must write to different locations, so the order that multiple accesses occur does not change the memory results. If this is not done, the memory contents of the accessed locations are undefined. For mutually exclusive IFc subgroups in a VLES, this rule applies independently to each subgroup.

Example 7-79. Pre-Calculated Memory Accesses to the Same Location

<code>move.w (r0+\$3),d0</code>	<code>move.l d1,(r0)</code>	<code>;not allowed</code>
<code>move.b (r0+n0),d0</code>	<code>move.l d1,(r0)</code>	<code>;not allowed if addresses overlap</code>
<code>move.w (r0+\$6),d0</code>	<code>move.b d1,(r1)</code>	<code>;not allowed if addresses overlap</code>
<code>bmset.w #\$123,(sp+\$10)</code>	<code>move.w d0,(r0)</code>	<code>;not allowed if addresses overlap</code>
<code>pop d0</code>	<code>move.l d1,(r0)</code>	<code>;not allowed if addresses overlap</code> <code>; and shadow SP is not valid</code>
<code>rts</code>	<code>move.w d0,(r0)</code>	<code>;not allowed if addresses overlap</code> <code>; and shadow SP is not valid</code>

Rule D.7

Instructions in the delay slot of delayed return instructions (RTED, RTSD, RTSTKD) cannot write to a memory address in the range $(SP-8 \leq \text{address} < SP)$, where SP is the value of the active stack pointer register at the beginning of the VLES having the RTED, RTSD or RTSTKD instruction.

Example 7-80. Memory Write to Stack in a Return Delay Slot

<code>tfra sp,r0</code>		
<code>move.l d1,(r0-8)</code>		<code>;allowed</code>
<code>rtsd</code>		
<code>move.l d2,(r0-8)</code>		<code>;not allowed</code>



7.6.3 RAS Rules

Rule J.4

Upon execution of the RTS or RTSD instruction, if the RAS is valid, the value of the RAS (used to restore the PC) must be equal to the value in the stack, pointed to by the SP, that would have been used if the RAS was not valid.

The following case manipulates the SP and builds on the return address using RAS:

Example 7-81. Illegal use of RAS value

```

move.l d6,(sp-8) adda #<8,sp
rts move.l d7,(sp-$c)           ; not allowed since RAS may be valid

```

7.6.4 Loop Rules

The loop COF instructions produce undefined results if all loops are disabled. Since the assembler cannot know the LFn state when these instructions execute, the simulator detects this programming rule. Good loop programming practices can ensure that rule L.N.6 is enforced.

Rule L.N.6

Loop COF instructions (BREAK, CONT, CONTD, and SKIPLS) cannot be used if all loops are disabled.

7.6.5 Rule Detection Across COF Boundaries

Some sequencing rules may be violated across COF boundaries — between instructions that are before the COF instruction or grouped in a VLES with the COF instruction (or its delay slot), and instructions at or after the COF destination. The assembler does not analyze this control flow and lacks the run-time execution trace to detect these cases. In addition, the code segments containing the COF instruction and the COF destination may be in two independently assembled source files, thus outside the visibility of the assembler. However, the simulator can detect most rule violations across COF boundaries by examining the simulation trace.

For example, the assembler cannot detect SR.2 across a COF boundary, but the simulator can detect it from the simulation trace.

Example 7-82. SR.2 Across a COF Boundary

```

    pop  sr
    bra  fred
    ...
fred
    add  d1,d2,d3           ;not allowed by SR.2
  
```

The assembler cannot detect the A.2 violation between the delay slot and the COF destination shown below, but the simulator can detect it from the simulation trace.

Example 7-83. A.2 from a Delay Slot to a COF Destination

```

    jmpd _dest
    bmsr #3,r0.1           ;2-cycle instruction
    ...
_dest
    move.l (r0),d0         ;not allowed by A.2
  
```

7.6.5.1 Cycle-Based COF Rules

Cycle-based COF rules calculate the COF cycle count to determine if the sequence is allowed or not. Cycle-based rule violations across COF boundaries are relatively rare because the COF instruction, taking multiple cycles, creates a “cycle barrier” between the VLES having the COF instruction and the VLES at the COF destination. However, this cycle barrier is reduced when the instructions are grouped in a VLES with the COF instruction (or its delay slot).

The relevant cycle-based rules that the simulator detects across COF boundaries are:

- A.1
- A.2
- A.2a

7.6.5.2 VLES-Based COF Rules

VLES-based COF rules are detected like static rules, except the rule is detected from the simulation trace, not the source code order. A VLES having a COF instruction is counted like any other VLES. Exceptions to the above are JSR/D and BSR/D instructions, guaranteeing an inherent one VLES distance between the instructions in the source execution flow (including the delay slot), and the instructions in the destination flow.

The relevant VLES-based rules that the simulator detects across COF boundaries are:

- Non-loop COF rules
 - T.1
 - SR.2
 - SR.4
- Loop COF rules
 - L.L.5
 - L.L.6
 - L.D.1
 - L.D.2
 - L.D.3
 - L.D.5
 - L.D.6
 - L.D.7
 - L.D.8
 - L.D.9
 - L.C.10
 - L.G.3
 - L.G.4

Example 7-84. Set condition during a COF, and use it at the destination (T.1)

```

    bt _des1cmpeq d0,d1
    ...
_des1 ift tfra r0,r1           ;Not allowed

    btd _des2cmpeq d0,d1
    nop
    ...
_des2 ift tfra r0,r1           ; Allowed

    bt _des3
    cmpeq d0,d1
    ...
_des3 ift tfra r0,r1           ; Not allowed

    jsrd _des4
    cmpeq d0,d1
    ...
_des4 ift tfra r0,r1           ; Allowed (1 extra VLES calculated between cmpeq and
                                ift)

```

7.6.6 Rule Detection Across Exception Boundaries

The SC140 can take an exception at most VLES boundaries in a program, and return using a RTE//RTED/ instruction after completing the exception service routine. The programming rules ensure that the transition from the program to the exception service routine, and the return back to the program are correct. The return back to the program is covered by the RTE//RTED/ programming rules in this chapter.

The following rules cover the transition from the program to the exception service routine. They apply to the first 1-2 VLES of the exception service routine, and do not apply to the program that was interrupted:

Rule SR.2a

Instructions that are affected by the C, T, S0, S1, or VF0-VF3 status bits in SR are not allowed in the first two VLES of an exception service routine. This rule does not apply to instructions that are affected by the EXP status bit in SR.

The assembler-mapped instruction CLR Dn is never affected by the SR status bits, even though it is implemented as SUB Da, Da, Dn. Therefore, this rule applies to the SUB instruction, but not to CLR (SUB Da, Da, Dn is taken as CLR in this context).

Rule SR.4b

MOVE-like instructions that reads or writes the EMR register are not allowed at the first two VLES of an exception service routine. The exception to this rule is any bit mask instruction(s) that does not have a zero value to the corresponding position of DOVF bit in the mask or that accesses the high portion of the EMR.

Example 7-85. EMR access at the start of an exception

```

; ISR Start
move.l emr.l, d0           ; Not allowed

; ISR Start
bmclr #$0004, emr.l       ; Not allowed

; ISR Start
bmclr #$fff0, emr.l       ; Allowed

; ISR Start
bmtstc #$0020, emr.l      ; Allowed

; ISR Start
move #$00000004, sr       ; Not allowed

```

Rule SR.6

The following instructions are not allowed in the first two VLES of an exception service routine:

- DOENn/DOENSHn
- CONT/CONTD
- BREAK
- SKIPLS

Rule A.1a

AGU instructions that read the R0-R7 registers with an address register update or address pre-calculation, or as an operand affected by a MCTL modifier field are not allowed at the first 2 cycles of an exception service routine. This rule does not apply to R8-R15, or to R0-R7 using the no update (Rn) addressing mode.

Example 7-86. MCTL Write to R0-R7 Use

```

; ISR Start
move.w (r0)+,d0           ;use MCTL, not allowed

; ISR Start
nop
move.w (r0)+,d0           ;use MCTL, not allowed

; ISR Start
nop
nop
move.w (r0)+,d0           ;use MCTL, allowed

; ISR Start
adda r0,r1                 ;use MCTL, not allowed

; ISR Start
move.w d1,(r0+n0)         ;use MCTL, not allowed

; ISR Start
move.w (r0)+,d0           ;use MCTL, not allowed

; ISR Start
move.w (r1)+,d0           ;use MCTL, not allowed

; ISR Start
move.w (r5)+,d0           ;use MCTL, not allowed

; ISR Start
adda r8,r1                 ;use MCTL, not allowed

; ISR Start
adda r1,r8                 ;no modifier mode, allowed

; ISR Start
adda #$1234,r8,r1         ;use MCTL, not allowed

; ISR Start
adda #$1234,r1,r8         ;no modifier mode, allowed

; ISR Start
move.w (r0),d0            ;no update mode, allowed

; ISR Start
move.w (r8)+,d0           ;no modifier mode, allowed

```

7.7 Programming Guidelines

The rules in this section cannot be detected within the visibility of the assembler and simulator. For example, the assembler and simulator cannot determine if the computed JMP below has a valid COF destination.

Example 7-87. Invalid COF Destination Cannot be Detected

```

    jmp r0                                ;assembler and simulator cannot determine
                                           ; if COF destination is the start of a VLES
  
```

The following rules must be detected by the programmer, and can be avoided by good programming practices. In addition to these rules, some good programming practices are presented below to assist the programmer in writing robust, compatible SC140 code.

Rule J.1

A COF destination must be the start (lowest) address of a VLES. A COF destination cannot jump into the middle of a VLES. This rule applies to all types of COF destinations described in [Section 7.4.8](#), “Change-Of-Flow Destinations.”

Example 7-88. COF Destination in the Middle of a VLES

```

    jmp _dest+2                            ;not allowed
    ...
_dest mac d0,d1,d2          mac d3,d4,d5
  
```

The assembler evaluates the address of a VLES label as the start (lowest) address of a VLES, regardless of its source position in the VLES. Good programming practice always places COF destination labels before or at the start of a VLES. Programmers should be careful that computed COF destinations are the start of a VLES. This will ensure that rule J.1 is enforced.

Rule J.2

A COF destination cannot be a delay slot.

Example 7-89. COF Destination in a Delay Slot

```

    move.l #_dest,r0
    ...
    jmp r0                                ;not allowed - COF delay slot
    jmpd _dest2
_dest
    add d0,d1,d2
  
```

Good programming practice never places COF destination labels before or inside a delay slot VLES. Programmers should be careful that computed COF destinations are not a delay slot. This will ensure that rule J.2 is enforced.

Rule J.3

Code should not be written to rely on the fact that delayed COF instructions and their delay slots are non-interruptible. In order to create non-interruptible sequences, the user should use other mechanisms such as the BMTSET.W instruction or encapsulating the code with the DI and EI instructions. Complying with this rule will help to insure compatibility with future StarCore architectures.

Rule J.5

A program section that ends near a border of reserved memory must end with a non-conditional change of flow (COF) instruction(s) followed by 4 aligned fetch sets (64 bytes) of NOPs or un-allocated memory. If the last meaningful instruction is not aligned with an end of a fetch set, some more NOPs are needed (up to 7) before 4 mentioned fetch sets. This is needed so that memory systems that generate an exception when reserved memory is accessed will not get an exception due to the pre-fetching of the core into the reserved memory.

Rule L.N.5

The LFn status bit (and SLF for a short loop) in SR must be set when executing loop body n (between the LOOPSTARTn and LOOPENDn directives).

Example 7-90. LFn Enabled During Loop Body n

```

dosetup1 label1
doen1 #5
pop sr                ;not allowed if pop disables LF1
...
loopstart1
label1
nop
nop
nop
loopend1

```

Good programming practice uses the loop control instructions listed in [Table A-14: AGU Loop Control \(Including Loop COF\) Instructions](#) on page A-18 for enabling and terminating loops. The programmer should not change the LFn and SLF status bits in SR while a loop is enabled. This will ensure that rule L.N.5 is enforced.

7.7.1 Rules Not Detected Across COF Boundaries

The simulator cannot detect all rules across COF boundaries. This may be due to limited analysis of the execution trace, or missing information in the binary encoding input to the simulator. The following rules must be detected by the programmer across COF boundaries:

- L.G.5

7.7.2 Good Programming Practices

Good programming practices assist the assembler and simulator in detecting programming rule violations. They also help the programmer write robust, compatible SC140 code. Some are generic to all software, and others are specific to the SC140. They are organized in several categories.

7.7.2.1 Source Code Practices

- Use symbolic COF destination labels and symbol arithmetic (not absolute addresses). Let the assembler resolve the label to a COF destination address.
- Use symbolic data labels (not absolute addresses). Use symbolic labels and symbol arithmetic for offsets into data structures. Let the assembler resolve the label to a data element address.

- Observe the immediate operand ranges as specified within the braces { } in [Appendix A.2, “Instructions,”](#) on page A-19. Operand values outside these ranges are undefined. Some specific examples are:
 - `ADD #u5, Dn { 0 ≤ u5 < 32 }`
 - `ASLL Da, Dn { -40 ≤ Da[6:0] ≤ 40 }`
- Observe address pointer alignments on memory accesses as specified in [Table 2-19: Memory Address Alignment](#) and within the braces { } in [Appendix A.2, “Instructions.”](#) Misaligned memory accesses are undefined. Some specific examples are:
 - `MOVE.L (a32), DR { 0 ≤ a32 < 232, L }`, meaning long word aligned (address is a multiple of 4).
 - `MOVE.4F (EA), Da:Db:Dc:Dd { 0 ≤ EA < 232, Q }`, meaning quad word aligned (address is a multiple of 8).
- Observe SP and OSP stack pointer quad alignment. The ESP and NSP registers have their three least significant bits hard-wired to zero. So SP arithmetic results that are not a multiple of 8 cannot be represented in the ESP and NSP registers. Note that MOVE-like instructions using SP with an offset to access the stack allow more general alignment based on the memory access size.
- Observe the word alignment requirement for COF destinations. Computed COF destinations (such as JMP R0) that are not a multiple of 2 are in error. Misaligned program fetches are undefined. Note that the PC register has its least significant bit hard-wired to zero.
- Observe modifier mode parameter constraints on size and offset discussed in [Section 2.3.4, “Address Modifier Modes.”](#) Using out-of-range or reserved parameter values may produce undefined results. Also observe address pointer alignments specific to the selected modifier mode.
- MOVES should be preceded by an instruction that updates the Ln-bit based on the data. Otherwise, the data moved may be modified by a Ln-bit not associated with the data.
- Do not explicitly modify the SR register to change the loop flags LFn and SLF in SR. Use the loop control instructions in [Table A-14: AGU Loop Control \(Including Loop COF\) Instructions](#) on page A-18.
- The SR register contains local task context such as the loop flags, T bit, etc. Always save and restore the SR register on exceptions and OS context switches.
- The EMR register contains global, not local, status bits. Do not use EMR status bits for local task context. Do not save and restore EMR on exceptions and OS context switches.
- Do not return from a subroutine with RTE/D, and do not return from an exception with RTS/D.
- If the return address on the memory stack is changed to effect a task switch, use RTSTK to bypass the RAS mechanism. Otherwise, the SC140 will return to the previous caller location if RAS is valid.
- Do not use VSL for anything other than the Viterbi algorithm. It violates the endian rules.
- Do not use “reserved” bits in registers for data storage. Always write “reserved” bits with their initial reset value (usually zero). This maintains software compatibility if a “reserved” bit is defined in the future.

7.7.2.2 Binary Code Practices

- Do not write self-modifying code (replacing portions of an application binary at run-time). It cannot be checked for errors by the assembler. It is also difficult to debug, and may not be compatible with future processor implementations.

- Do not write explicit binary encodings using DC (declare constant) assembler directives. It cannot be checked for errors by the assembler.
- Do not use “reserved” or “--” operand field values in instruction encodings. This maintains software compatibility if a “reserved” or “--” field is defined in the future.

7.7.2.3 Software Development Practices

- Programmers should not disable programming rule detection by the assembler and simulator.
- Programmers should have an application test suite that provides good dynamic rule test coverage using the simulator.
- The assembler cannot detect rule violations across source file boundaries. Partition the application program into separate source files at logical points, using COF instructions to pass program control between source files to avoid possible rule violations.
- The linker does not detect programming rules. Do not link several object files into sequential binary code. Use COF instructions to pass program control between linked object files to avoid possible rule violations.
- Static detection of hardware loop iterations (LA to SA sequences) requires all of loop body n including the DOENN/DOENSHn instructions, LOOPSTARTn and LOOPENDn directives to be in the same source file. Follow the static detection assumptions given in [Section 7.5.1, “Hardware Loop Detection,”](#) on page 7-7.
- Do not reassemble disassembled code. The disassembled code may contain hardware loop LPMARKx prefix instructions that are not supported in SC140 source code. Use one LOOPSTARTn and LOOPENDn directive to mark the hardware loop body n in the assembly source code.
- Write endian-independent code wherever possible. Document code that is endian-specific.
- Document status bit assumptions (such as 32-bit arithmetic saturation mode, SR[SM]) in application programs.

7.8 LPMARK Rules

The SC140 encodes LPMARK bits in the first prefix word to mark the end of hardware loops. The LPMARK bits are automatically encoded by the assembler based on the LOOPSTARTn and LOOPENDn assembly directives. The encoding procedure is defined in the LPMARKx (LPMARKA and LPMARKB) instruction definitions in [Appendix A.2, “Instructions.”](#)

Generally, the LPMARKA and LPMARKB instructions are disassembler syntax only for host debugging. The assembler flags their use in source code as a fatal error. However, the LPMARKA and LPMARKB instructions can be used in source code for hardware testing, enabled by an assembler switch.

The LPMARK rules in this section are defined for two purposes - 1) how to construct correct test code using the LPMARKA and LPMARKB instructions, and 2) how the LPMARK bits encoded in the prefix are used by the simulator to detect some SC140 programming rules.

7.8.1 LPMARK Instruction Type

LPMARK is classified as a prefix instruction type for all SC140 programming rules.



7.8.2 Static Programming Rules

This section defines new SC140 LPMARK programming rules for correct LPMARKA and LPMARKB instruction use in a VLES, when enabled by an assembler switch. When enabled, these rules apply in addition to the other programming rules.

7.8.2.1 General Grouping Rules

LPMARK Rule G.G.1

The LPMARKA and LPMARKB instructions are not counted for this rule.

7.8.2.2 Prefix Grouping Rules

LPMARK Rule G.P.3

Multiple LPMARKA instructions cannot be grouped in a VLES. Multiple LPMARKB instructions cannot be grouped in a VLES. This LPMARK rule applies to the whole VLES.

LPMARK Rule G.P.6

The LPMARKA and LPMARKB instructions are not counted for this rule.

LPMARK Rule G.P.7

IFc instructions do not affect the LPMARKA and LPMARKB instructions. The LPMARKA and LPMARKB instructions always execute unconditionally, and can be placed anywhere in the assembly source order.

7.8.3 Dynamic Programming Rules

The LPMARK rules in this section are alternate forms of SC140 programming rules detectable from the prefix encoding. Source code that complies with the assembly notation rules is by definition compliant with LPMARK rules. These LPMARK rules allow the simulator to detect dynamic programming rules that are not detectable by the assembler.

7.8.3.1 LPMARK Notation

The LPMARK rules use the VLES address notation “LPA-2, LPA-1, and LPA” (relative to the VLES having LPMARKA set) and “LPB, LPB+1 and LPB+2” (relative to the VLES having LPMARKB set). The minus “-” notation adjusts the VLES address earlier in the object code order.

7.8.3.1.1 Active Loop

In a nested loop structure, more than one loop can be enabled at the same time. A loop is enabled when its LFn bit in SR is set, where n is the loop index. The enabled loop with the highest index is defined as the “active loop”. This definition is dynamic and follows the SC140 loop state machine. The SC140 loop state machine and simulator determine the “active loop” from the LFn bits in SR when a VLES having an LPMARK bit set is executed.

7.8.3.1.2 Active SAn Register

“Active SAn register” is defined as the SAn register where n = the active loop index. This definition is dynamic and follows the SC140 loop state machine.

7.8.3.1.3 Active LCn Register

“Active LCn register” is defined as the LCn register where n = the active loop index. This definition is dynamic and follows the SC140 loop state machine.

7.8.3.2 Loop Nesting Rules

LPMARK Rule L.N.5

At least one LFn status bit in SR must be set at LPA or LPB of a loop.

Example 7-91. LFn Enabled at LPA or LPB

```

dosetup1 label1
doen1 #5
pop sr                ;pop clears all LFn
...

label1
nop                   {lpmarkb set}    ;not allowed
nop
nop
    
```

7.8.3.3 Loop LA Rules

LPMARK Rule L.L.1

The following instructions are not allowed at LPB+1 or LPB+2 of a long loop:

- COF instructions
- STOP and WAIT
- DI
- DEBUG

Example 7-92. Instructions at the End of Long Loops

```

move.w #count2,d6
dosetup0 label2
doen0 d6
move.w #1,d1
move.w #2,d2
move.w #3,d3
move.w #4,d4

label2 inc d1
      inc d2
      inc d3      {lpmarkb set}
      inc d4
      wait                ;not allowed
    
```

LPMARK Rule L.L.2

A DOENn or MOVE-like instruction that writes the active LCn register is not allowed at LPB, LPB+1, or LPB+2 of a long loop.

Example 7-93. Active LCn Write at the End of Long Loops

```
doen1 #3
move.l #$12345678,r1
inc d0      {lpmarkb set}
doen1 #5                    ;not allowed
move.w  d3,(r1)+
```

LPMARK Rule L.L.3

The following instructions are not allowed at LPA, LPB, or LPB+1 of a short loop:

- COF instructions
- STOP and WAIT
- DI
- DEBUG
- DOENn/DOENSHn
- MOVE-like instructions that read any LCn register
- MOVE-like instructions that write any LCn register
- MOVE-like instructions that read the SR register
- MOVE-like instructions that write the SR register

This rule does not apply to other instructions that affect status bits in SR.

Example 7-94. Instructions in Short Loops

```
move.w  #count2,r6
doensh0 r6
move.w  #3,d3

inc d3      {lpmarkb set}
wait                    ;not allowed

doensh0 #$10
nop

jmp end    {lpmarka set}    ;not allowed

doensh1 #count2
move.w  #num,d2

doen1 #5    {lpmarka set}    ;not allowed
```

LPMARK Rule L.L.5

A MOVE-like instruction that writes the SR register is not allowed at LPB-2, LPB-1, LPB, LPB+1, or LPB+2 of a long loop.

LPMARK Rule L.L.6

A MOVE-like instruction that writes the SR register is not allowed at LPA-2, LPA-1, LPB-2 or LPB-1 of a short loop.

7.8.3.4 Loop Sequencing Rules

LPMARK Rule L.D.2 + L.D.3

The minimum number of VLES between the following instructions that write the active LCn register and LPA or LPB of a loop is:

- DOENn/DOENSHn Rn or #x: one VLES (address register or immediate value)
- DOENn/DOENSHn Dn: two VLES (data register)
- MOVE-like instruction that writes the active LCn register: two VLES

Example 7-95. Active LCn Write at the Start of a Loop

```

move.w #3,d0
doensh0 d0           ;allowed
move.l d1,lc0       ;not allowed
move.w #2,d2

inc d1      {lpmarka set}

move.w #3,r8
dosetup1 label1
doen1 r8           ;not allowed

label1 inc d3      {lpmarkb set}
       inc d4
       inc d5

```

LPMARK Rule L.D.6

At least one VLES is required between an instruction that writes the active SAn register and LPA or LPB of a long loop.

Example 7-96. Active SAn Write at the End of Long Loops

```

doen0 #5
dosetup0 label           ;not allowed
label inc d2      {lpmarkb set}
       inc d1
       inc d0

```

LPMARK Rule L.D.8 + L.D.9

At least one VLES is required between a MOVE-like instruction that reads the active LCn register and LPA or LPB of a loop.

Example 7-97. Active LCn Read at the Start of a Loop

```

doensh0 # $10
push  lc0                ;not allowed

inc  d0    {lpmarka set}

doensh0 # $10
push  lc1                ;allowed

inc  d0    {lpmarka set}

move.w #count2,d6
dosetup0 label2
doen0  d6
move.w #1,d1
move.w #2,d2
move.w #3,d3
move.w #4,d4

label2 inc d1
move.l lc0,d0            ;not allowed
inc d2    {lpmarkb set}
inc d3
inc d4

move.w #count2,d6
dosetup0 label2
doen0  d6
move.w #1,d1
move.w #2,d2
move.w #3,d3
move.w #4,d4

label2 inc d1
move.l lc1,d0            ;allowed
inc d2    {lpmarkb set}
inc d3
inc d4
    
```

7.8.3.5 Loop COF Rules

LPMARK Rule L.C.2

COF instructions are not allowed at LPB of a long loop.

Example 7-98. COF Instructions at LPB of a Long Loop

```

dosetup1 label1
doen1 #n2
move.l #mem_l1,r1
move.l #mem_l2,r0

label1 inc d1
      jsr r1      {lpmarkb set}      ;not allowed
      add d1,d2,d3
      move.w d3,(r0)
    
```

LPMARK Rule L.C.3 + L.C.5

A Bc or Jc instruction is not allowed at LPA-1 or LPB-1 of a loop.

Example 7-99. Bc/Jc at the Start of a Loop

```

      cmpgt d4,d3
      nop
      iff doensh3 #count2
      bt _dest      ;not allowed

      inc d2      {lpmarka set}
      ...
_dest inc d2

dosetup1 label7
move.w #0,d1
doen1 #5
move.w #10,d2

label7 inc d1
      bf label6      ;not allowed
      inc d2      {lpmarkb set}
      inc d3
      inc d4
    
```

LPMARK Rule L.C.9

A loop COF instruction (BREAK, CONT, CONTD, or SKIPLS) cannot have a COF destination that is at LPB of a long loop if immediately followed by LPA.

Example 7-100. Loop COF at End of Nested Long Loops

```

doen0 #5
...
doen1 #10
...
doen2 d0
nop
skipls _dest                ;not allowed
nop
ift break label
nop
nop          {lpmarkb set}
nop
nop
...
nop          {lpmarkb set}
_dest
nop          {lpmarkb set}
label nop    {lpmarka set}    ;last address of long loop 1
nop          ;last address of long loop 0
    
```

LPMARK Rule L.C.10

A BSR, BSRD, JSR, or JSRD instruction cannot have a COF destination that is at LPA or LPB of a loop.

Example 7-101. Subroutine Call to End of Loops

```

dosetup0 label1
doen0 d1
nop
nop

label1 nop
nop
jsr label2                ;not allowed
nop
nop
inca r1
label2 inca r7          {lpmarkb set};LA-2
add d1,d2,d3           ;LA-1
move.w d3,(r0)         ;LA
    
```



LPMARK Rule L.C.11 + L.C.12

A delayed COF instruction is not allowed at LPA-1 or LPB-1 of a loop.

Example 7-102. Delay Slot at LPA or LPB of a Loop

```

    jmpd _dest                ;not allowed
    nop                      {lpmarkb set}
    nop
    nop

    jmpd _dest                ;not allowed
    nop                      {lpmarka set}

```

7.8.3.6 General Looping Rules

LPMARK Rule L.G.3 + L.G.4

At least one VLES is required between a MOVE-like instruction that reads the SR register and LPA or LPB of a loop.

Example 7-103. SR Read to LPA or LPB of a Loop

```

    dosetup1 label1
    doen1    #5
label1
    inc d1
    move.l sr,d0                ;not allowed
    inc d2    {lpmarkb set}
    move.l #mem_11,r1
    move.l #mem_12,r0

    doensh0 #$10
    push sr                    ;not allowed

    inc d0    {lpmarka set}

```

7.8.3.7 Rule Detection Across Exception Boundaries

LPMARK Rule SR.6

LPA or LPB cannot be the first two VLES of an exception service routine.

7.8.4 LPMARK Programming Guidelines

The rules in this section cannot be detected by the simulator from its execution trace. The following rules must be detected by the programmer, and can be avoided by good programming practices.

LPMARK Rule L.C.1

A COF instruction cannot have a COF destination that is LPB+1 or LPB+2 of a long loop or LPB+1 of a short loop. This rule does not apply to loop COF instructions (BREAK, CONT, CONTD and SKIPLS) in a nested loop having a COF destination that is LPA-1 or LPA of an enveloping loop.

Example 7-104. COF Destination to Loop Delay Slots

```

doenshl #5
...
cmpeq.w #3,d0
jfb _dest ;not allowed
inc d0

inc d0 {lpmarkb set}
_dest add d1,d2,d3
  
```

Good programming practice never places COF destination labels before or inside a loop delay slot VLES, unless the label is the destination of a loop COF instruction as described above. Programmers should be careful that computed COF destinations are not a loop delay slot. This will ensure that LPMARK rule L.C.1 is enforced.

7.9 NOP Definition

Programmers use NOP as a **deterministic** way to control word padding within a VLES, and cycle padding in a program. The architecture definition of NOP and its assembler encoding follows:

1. A “**baseline VLES**” is defined as the source code and binary encoding of the VLES without NOPs in the source code. The baseline VLES has a “**baseline size**” in words (“**W**”, a 16-bit unit). It may or may not include prefixes, reordered encoding, or modulo alignment padding. The “**baseline encoding**” may pad a word for modulo alignment, but it should not be called a NOP in the Tools document or the CRM. In this discussion, an assembler-generated NOP encoding not present in the source code will be called a “**PAD**”.
2. A “**NOP**” is defined in the CRM Appendix A as a source syntax having a 1W prefix encoding that can be used in a standalone NOP-only VLES or embedded in a baseline VLES having a prefix. The binary encoding of the standalone NOP is a 1W prefix having the VLES size in the aaa field, and the embedded NOP is a 1W prefix having aaa=0, as specified in the CRM Appendix. A. **The NOP definition is to increase the baseline size by 1 word for each source NOP added to the baseline VLES.** This means that NOPs are not compressed or absorbed into baseline VLES encodings already having prefixes. They are concatenated as higher addressed words with the baseline VLES encoding occupying the lower addresses.
3. Prefix grouped VLES already have a prefix, and each NOP adds one 1W embedded NOP prefix having aaa=0 to the baseline VLES encoding. Serially (non-prefixed) grouped VLES are encoded without a prefix, so the first NOP is added differently. The first NOP encodes as a standalone 1W prefix having the VLES size in the aaa field, and corrects the serial grouping field of the baseline instructions. Any additional NOPs are encoded as in prefix grouping.
4. When NOPs are used in a standalone VLES, the baseline size is zero. A VLES containing **only** N NOPs in the source code has a size of N words. It is implemented by a standalone 1W NOP prefix having the VLES size in the aaa field followed by N-1 embedded 1W NOP prefixes having aaa=0. This is the only guaranteed way to pad N consecutive words.

5. Source syntax order in a VLES generally has no effect on the baseline size, as parallel semantics define no instruction serialization within a VLES. The baseline size is determined by the encoding rules in the assembler. There are two cases where source order matters for the programmer - 1) for multiple conditional subgroups in a VLES, any IFA subgroup must be the last (right-most) subgroup in the source syntax, and 2) if multiple instructions in the same VLES affect SR[C], the last (right-most) instruction in the source syntax that actually executes updates SR[C]. This source order syntax affects the encoding order of the VLES and can affect the baseline size, but not the size increase due to NOPs in the source VLES.
6. Conditional (IFc) syntax requires special rules to provide deterministic NOP word padding when the two subgroups are used. If a conditional group or subgroup contains only NOPs, it should be assembler mapped as follows (a colon “:” is the concatenation operation):

IFT NOP --> IFT (prefix ccc=010):NOP

IFF NOP --> IFF (prefix ccc=011):NOP

IFT NOP IFA subgroup2 --> IFA (prefix ccc=000) group2:NOP (same as group2:NOP)

IFF NOP IFA subgroup2 --> IFA (prefix ccc=000) group2:NOP (same as group2:NOP)

IFT subgroup1 IFA NOP --> IFT (prefix ccc=010) group1:NOP

IFF subgroup1 IFA NOP --> IFF (prefix ccc=011) group1:NOP

IFT subgroup1 IFF NOP --> IFT (prefix ccc=010) group1:NOP

IFT NOP IFF subgroup2 --> IFF (prefix ccc=011) group2:NOP

This mapping converts each NOP-only conditional group or subgroup to embedded NOPs concatenated with a conditional group (no subgroups). This is necessary to ensure that each NOP adds exactly one word to the baseline size. Some of these are not useful cases, but are legal source syntax for the assembler.

If a conditional group or subgroup has a NOP with other instructions, the NOPs should ignore their subgroup when concatenating with the baseline VLES encoding. In other word, NOP padding can be added to either subgroup as shown in the examples.

7. The FALIGN assembler directive should assemble the VLES source code including existing source NOPs, determine the VLES address boundaries, and add the minimum number of PADs to achieve the desired fetch address alignment. The assembler can spread PADs across several VLES (including conditional VLES) to avoid a cycle penalty, or add them as a standalone VLES if the alignment cannot be achieved without a cycle penalty.

7.9.1 Grouping Examples

Assume there are N NOP instructions in the source VLES (including PADs added by the FALIGN directive).

1. If a baseline VLES does not require a prefix, the first source NOP is encoded as a 1W VLES prefix. For example:

[INC DONOP]

is encoded as:

[1W prefix, INC]

and

[2W IFT-IFF prefix, INC, CLR, NOP]

and

[IFF CLR D8IFT INC D1 IFT NOP]

is encoded (ignoring the NOP subgroup) as:

[2W IFT-IFF prefix, CLR, INC, NOP]

7. If a baseline VLES has multiple NOPs in a conditional subgroup, a 1W embedded NOP is encoded for each source NOP. For example:

[IFT CLR D0IFT NOP IFT NOP]

is encoded as:

[1W IFT prefix, CLR, NOP, NOP]



Appendix A

SC140 DSP Core Instruction Set

A.1 Introduction

This appendix describes in detail the SC140 instruction set, its encoding, and its syntax. The first pages of this appendix contain information common to all of the instructions such as the conventions, notation, and syntax used in the Appendix. Next, the encoding for the prefix words is given. Then, the names and simple descriptions of the instructions are listed in functional groups. At the end of the introductory section, a single page describes the format of the instruction descriptions, which are listed in alphabetical order through the bulk of the Appendix.

Each non-prefix instruction activates one functional unit in the SC140 architecture. The architecture can be viewed as several functional units operating in parallel:

- Four arithmetic logic units (ALUs)
- Two address arithmetic units (AAUs)
- One bit mask unit (BMU)
- One program controller (PSEQ)

Several instructions can be grouped together for parallel execution.

The instruction set has been designed to enable efficient parallel execution of DSP algorithms and control code, using high-level language compilers, achieving maximum speed and minimum power consumption. This extensive range of instruction capabilities also provides a very powerful assembly language for DSP algorithms and general-purpose computing. Certain programming rules apply regarding the ability to group instructions that activate the various units, because of their use of shared resources.



1.1 Conventions

Table A-1 lists the conventions used in this appendix to define the instructions.

Table A-1. Instruction Conventions

Convention	Definition
()	Indirect address
aa	Absolute address
Cn	Control registers
Da	Single source/destination data register
Da:Db	Source/destination data register pair
De.E; Do.E	Data register extension (bits 39:32 + Ln bit)
De.E:Do.E	Data register extension pair (e.g., D4.E:D5.E)
Db	Single source data register
De	Even numbered data/core register
Dn	Destination data register
Do	Odd numbered data/core register
DR	Data or address register
Ea	Effective address
HP	High portion (bits [31:16]) of a register
Ln	Limit tag bit
LP	Low portion (bits [15:0]) of a register
rc	Rounding constant
Rn	Address register
rx	AGU source register
Rx	AGU source/destination register
{ }	If used at the end of a line, the intent is merely for clarity purposes, and this is not part of the assembler syntax. However, note that if this is used in assembler code, the contents will be understood as ISAP instructions.
[b:a]	Bit range a to b in a register or memory
[a]	Bit number a in a register or memory

Table A-2 describes the operators and operations syntax for each instruction.

Table A-2. Operations Syntax

Operator	Description
+	Add
-	Subtract
*	Multiply
/	Divide
	Absolute value
&	Logical AND
	Logical OR
⊕	Exclusive OR
~	Bitwise complement
==	Test for equality, 1 if equal, 0 if not equal
→	Transfer left to right
↔	Indicates either right or left transfers, but not both at once
>>	Arithmetic right shift (sign bits shifted right)
<<	Arithmetic or logical left shift (functionally the same)
>>>	Logical right shift
>	Compare for greater than
Rnd()	Rounding function
x:y	Concatenation of x and y

Table A-3 describes the abbreviations used for the core registers.

Table A-3. Register Abbreviations

Abbreviation	Register Name
D0-D15	General purpose data register
R0-R15	General purpose address register
EMR	Exception and mode register
VBA	Vector base address register
SP	Stack pointer registers: normal (NSP) and exception (ESP)
PC	Program counter
SR	Status register
MCTL	Modifier control register
SA0-SA3	Start address registers
LC0-LC3	Loop counter registers
B0-B7	Modulo base registers
N0-N3	Offset registers
M0-M3	Modulo registers
OSP	Other stack pointer

Table A-4 lists special syntax used in this appendix to define an instruction's assembler syntax. Note that the assembler syntax is case insensitive.

Table A-4. Assembler Syntax

#	Prefix for an immediate value (for example, #u5 means an immediate 5-bit unsigned number and #s16 is an immediate 16-bit signed number).
\$	Prefix to a hexadecimal value (for example, #\$1A4F as an immediate value or \$2FC as an address offset).
>	Prefix for long addressing, forces the assembler to use an extra word to encode the displacement/offset.
<	Prefix for short addressing, forces the assembler to use the smallest instruction when encoding the displacement/offset.
label	Replace the word "label" in an instruction with the label name of an execution set in code. The instruction determines if the assembler substitutes an absolute address or a relative displacement in the opcode.
*	Assembler variable containing the address of the current execution set.
{ }	Used to include ISAP instructions, OR - i If the contents are not ISAP instruction, then these brackets are merely for illustrative purposes only, and can be used to define the range of addressing of the previous instruction

A.1.1.1 Brackets as ISAP indicators

The SC140 core can dispatch one *ISAP opcode* per VLES. This opcode uses the 2-word prefix encoding, and is recognized as an ISAP opcode if it is not the first opcode in the VLES. The assembly syntax uses brackets to identify a mnemonic as an ISAP instruction, for example:

```
ADD D0,D1,D2 {move_spatial (r0)+,K0}
```

For more details about working with an ISAP, see [Chapter 6, "Instruction Set Accelerator Plug-In."](#)

A.1.1.2 Brackets as address indicators

The ranges for addresses are included in the assembly syntax sections. These are not part of the assembly syntax, but are shown there for clarity. They are enclosed in curly brackets { }. For example, in MOVE.L the following is found:

```
MOVE.L (Rn+u3),DR {0 ≤u3 < 32,L}
```

In this example, u3 is an unsigned immediate offset to the value in pointer register Rn. In addition, u3 is aligned to long word addressing. Its values are in multiples of four. Addressing is in units of bytes, and there are four bytes to a 32-bit long word. With only every fourth address required, u3 is encoded with three bits to encode: 0, 4, 8, ..., 28. The notation for these numbers is $\{0 \leq u3 < 2^5, L\}$. The values for u3 in the source code and disassembled code will be multiples of four from 0 to 28. The 3-bit encoded values in the instruction will be zero to seven. Other notations for address alignment are:

- W for word (multiples of two)
- Q for quad (multiples of eight)

The ranges shown in the brackets are always for the source code addressing. The ranges may or may not reflect the number of bits used in the encoding, depending on whether a left shift is applied to the encoded value.



1.2 Addressing Mode Notation

Table A-5 and Table A-6 define the fields in the address offset or post increment tables, which are found in the instruction field section of an instruction definition. EA stands for effective address. A different field in the opcode (RRR) determines what register is used for Rn.

EA	MMM	Effective Address Notation					
000	(Rn+N0)	010	(Rn)	100	(Rn)+N0	110	(Rn)+N2
001	(Rn)–	011	(Rn)+	101	(Rn)+N1	111	(Rn)+N3

Table A-5. Addressing Mode Notation for the EA Operand

Addressing Mode Definition	Notation in the Instruction Field
Indexed by offset in N0	(Rn + N0)
Post decrement	(Rn)–
No update	(Rn)
Post increment	(Rn)+
Post update by offset in N0	(Rn) + N0
Post update by offset in N1	(Rn) + N1
Post update by offset in N2	(Rn) + N2
Post update by offset in N3	(Rn) + N3

Note: Rn is taken from the Rn (RRR) table found in the instruction definition.

ea	MM	Effective Address Notation					
00	(Rn)+	01	(Rn)–	10	(Rn+N0)	11	(Rn)

Table A-6. Addressing Mode Notation for the ea Operand

Addressing Mode Definition	Notation in Instruction Field
Indexed by offset in N0	(Rn + N0)
Post decrement	(Rn)–
No update	(Rn)
Post increment	(Rn)+

Note: Rn is taken from the Rn (RRR) table found in the instruction definition.

1.3 Data Representation in Memory for the Examples

For the examples in this appendix, the convention for the representation of data in memory is to show the same order in memory as is in the source register for a write. For example, a 32-bit write from a register containing \$12345678 to address \$100 will be shown as: (\$100) = \$12345678. The exact order of multi-byte operands in memory depends on the endian mode, and is described in [Section 2.4.1, “SC140 Endian Support,”](#) on page 2-56.

A.1.4 Encoding Notation

The instruction encoding is defined for each instruction under **Instruction Formats and Opcodes** and **Instruction Fields**. Each instruction field may not be contiguous in the opcode, but the order of the bits in each field is consistent from the opcode to the definition of the field. For example, in the MOVE.B instruction, the encoding for the opcode is

MOVE.B (a16),DR	15		8	7		0										
	0	0	0	1	H	H	H	H	A	A	A	0	1	1	1	0
	1	0	0	A	A	A	A	A	A	A	A	A	A	A	A	A

The order of bits for a16 (AAA---A) is the MSB in the first bit at the left-most position in word 1. The LSB is the right-most position in word 2. If written out fully, the encoding would be:

MOVE.B (a16),DR	15		8	7		0										
	0	0	0	1	H	H	H	H	A15	A14	A13	0	1	1	1	0
	1	0	0	A12	A11	A10	A9	A8	A7	A6	A5	A4	A3	A2	A1	A0

A more complicated example is for BSR, where two fields are intermixed. The order for each field is maintained, monotonically decreasing from left to right.

BSR >label	0	0	1	0	a	0	1	1	A	A	A	1	1	a	a	a
	1	0	0	A	A	A	A	A	A	A	A	A	A	A	a	a

The definition for the field is:

displacement aaaaaAAAAAAAAAAAAAAAAA0 20-bit signed PC relative displacement
(>label)

If the field was written out in the encoding table, it would appear as follows:

BSR >label	0	0	1	0	a20	0	1	1	A15	A14	A13	1	1	a19	a18	a17
	1	0	0	A12	A11	A10	A9	A8	A7	A6	A5	A4	A3	A2	A1	a16



1.5 Prefix Word Encoding

Each execution set can be associated with a one-word (low or high register) or two-word prefix that is placed at the beginning of the set. A prefix conveys additional information about the set such as:

- Conditional execution of an execution set or a subgroup (originating in the IFT/IFF/IFA instructions).
- The number of instructions that are grouped together in the execution set.
- Looping information to support hardware loops (lpmarkA and lpmarkB bits).
- Encoding extension for high register banks (D8–D15, R8–R15).

There are two prefix formats: a one-word low register prefix, and a two-word prefix. A one-word low register prefix encodes information concerning the first three items above. A two-word prefix includes information on all items above.

The basic 16-bit instruction encoding of the SC140 has three bits allocated to specify a data or pointer register. Therefore, these encodings alone can specify only eight DALU registers (D0–D7) and eight address pointers (R0–R7). In order to specify operands that belong to the high register banks (D8–D15, R8–R15), additional encoding bits are needed. These bits are allocated in the two-word prefix. A two-word prefix includes a field for each execution unit in the SC140: four fields for DALU instructions and two fields for AGU instructions.

DALU instructions have a maximum of three operands, so each DALU field is 3 bits wide. AGU instructions have a maximum of two operands, so each AGU field is two bits wide. This provides an encoding extension bit for each possible operand in each execution unit. If a bit is set, it signifies that the respective operand uses a high-bank register. If the associated bit is clear, the operand uses a register from the low bank. A two-word prefix is generated by the assembler if at least one of the instructions in the execution set that uses a register from the high banks has 3 operands, and/or is conditionally executed or is in a loop.

Prefix words are optional, generated by the assembler if needed. The rules used by the assembler to determine if a prefix is needed are described in [Section 5.2.4, “Prefix Selection Algorithm.”](#) :

Note: Use of a prefix reduces the space available for instructions in the eight-word execution set by the size of the prefix. For example, if an instruction that references a high-bank register causes the assembler to generate a two-word prefix, only six words are left available in that execution set for instructions.

Example:

```

        skip1 _last           ;(there is a skip1 to _last in the program)
        .
        .
        .
        execution_set
        execution_set
        _last           execution_setlpmarkA
    
```

In the case of the loop having just one execution, the lpmarkA bit is set in the prefix of this single execution set of the loop.

A.1.5.2 Two-Word Prefix

Includes information on grouping, looping, IFc (conditional execution), and high-register banks (D8-D15, R8-R15).

Instruction Formats and Opcodes

Prefix	Words	Cycles	Type	Opcode												
2W PREFIX	2	0	4	<table border="1"> <tr> <td style="text-align: center;">15</td> <td style="text-align: center;">8</td> <td style="text-align: center;">7</td> <td style="text-align: center;">0</td> </tr> <tr> <td>0 0 1 1 a a a 0</td> <td>H t h p j c c c</td> <td></td> <td></td> </tr> <tr> <td>1 0 1 b B e E T</td> <td>b B e E b B e E</td> <td></td> <td></td> </tr> </table>	15	8	7	0	0 0 1 1 a a a 0	H t h p j c c c			1 0 1 b B e E T	b B e E b B e E		
15	8	7	0													
0 0 1 1 a a a 0	H t h p j c c c															
1 0 1 b B e E T	b B e E b B e E															

Note: The order of the register bank encoding fields is, for example, E1 E2 E3, with E1 occupying the most significant bit position in the table.

Instruction Fields

aaa: Number of instruction words being grouped, including the prefix word minus one (for example, 2-w prefix + 2 grouped instruction words, aaa = 3).

For a 2-w prefix at the beginning of the execution set, aaa = 000 and aaa = 001 are reserved as escape codes to signify that more prefix words are concatenated to support architectures with 3 or more prefix words. Use of a 2-w prefix in the middle of the set is reserved for future encoding (such as accelerator or predication instructions) and should not be placed as a NOP.

ccc: Conditional execution of the entire execution set.

In the following table, true/false relates to the state of the T bit in SR. D0, D1, D2, and D3 are DALU instructions, A0 and A1 are AGU or BMU instructions. The numbers relate to the relative offset of the instruction from the beginning of the set.

000—Unconditionally executed
 001—If true (D0, D2, A0), if false (D1, D3, A1)
 010—If true, all the set
 011—If false, all the set
 100—Reserved
 101—Reserved
 110—If true (D0, D2, A0), always (D1, D3, A1)
 111—If false (D0, D2, A0), always (D1, D3, A1)

p: lpmarkB bit

In the case of a loop with three or more execution sets, the lpmarkB bit is a one in the execution set that is two before the last execution set in the loop.

Example:

```

        lpmarkB; (set LA - 2)
                ; (set LA - 1)
    _last      ; (set LA)
    
```

In the case of a loop with two execution sets such as SA mark, the lpmarkB bit is set in the first execution set of the loop.

Example:

```

    _start execution_set lpmarkB
    _last  execution_set      ;(this is a loop on two execution sets)
    
```

j: lpmarkA bit

In case of a loop with more than three execution sets, the lpmarkA bit is set in the prefix of the execution set, which is at _last only if there were any SKIPLS, BREAK, CONT, CONTD to _last, or to _last-1.

Example:

```

        skipl _last      ;(there is a skipl to _last in the program)
        .
        .
        .
    execution_set
    execution_set
    _last      execution_set lpmarkA
    
```

In the case of the loop having just one execution set, the lpmarkA bit is set in the prefix of the first execution set of the loop.

EEE: Data register expansion for DALU execution unit 0 (bit names: E1, E2, E3). This includes all DALU instructions. For three-operand instructions (inst op1,op2,op3):

--1: high data register is used for the op3 field (E3 is set)
 -1-: high data register is used for the op1 field (E2 is set)
 1--: high data register is used for the op2 field (E1 is set)

In case of two-operand MAC unit instructions, only E1 and E3 are used. In case of one-operand MAC unit instructions, only --E3 is used.

In case of four-operand instructions with data registers defined as third and fourth operands, E1 and E3 used similarly to two-operand instructions.

Each of the E bits serves as a fourth register field encoding bit, effectively turning FFF → FFFF, etc. For example, the D0-7 registers are expanded to D0-15. Each E bit selects the low or high bank of registers for that field.

Fields representing multiple registers in DALU instructions could be scaled up independantly. For instance, the register pair encoding in the instruction MAX D0,D4 can be expanded to MAX D8,D4.

eee: The same as EEE, but for DALU execution unit 1.

BBB: The same as EEE, but for DALU execution unit 2.

bbb: The same as EEE, but for DALU execution unit 3.



1: High register expansion encoding for AGU execution unit 0. This includes all AGU and BMU instructions. R0-7 registers are expanded up to R0-15.

The two bits Hh controls the expansion of R0-R7 registers to R8-R15 registers in one or two AGU instruction's operands according to the following rules:

The H bit is used for all of the operands from the types:

- Rn operand defined with RRR field (e.g. Rn in MOVE like instruction, or ADDA instructions, EA or ea in MOVE like instructions).
- Rx operand defined with RRRR field.
- Df operand defined with hhh field of MOVE.L Df,C4 and MOVE.L C4.Df instructions.

The h bit is used for all the operands other than the above.

The expansion encoding have no effect if the register decoded in the instruction is not R0-R7 (e.g. SP in RRRR field for Rx decoding).

Fields representing multiple registers (used in some MOVE-like instructions) are affected together. For example, the register pair encoding for D0:D1 can be expanded to D8:D9 (not each register independently).

Note that Rr register in (Rn+Rr) addressing mode, is limited to R0-R7 and thus not effected by both H and h bits.

Tt: The same as Hh, but for AGU execution unit 1.

Note the special position of the T bit (bit 8 in the second word).

1.6 Instruction Types

The SC140 instruction set is organized into the following instruction types, specified at the top of every instruction definition in this Appendix:

DALU Instructions- perform operations on the data registers D0-D15 using the DALU execution units (MAC and BFU). All DALU instructions are listed in Table A-7 and Table A-8. The architecture is described in [Section 2.2.1, “DALU Architecture,”](#) on page 2-6.

AGU Instructions- perform operations using the AGU execution units (AAU and BMU) and the program sequencer unit. All the AGU instructions are listed in Table A-9 through Table A-15. The architecture is described in [Section 2.3.1, “AGU Architecture,”](#) on page 2-31.

BMU Instructions - are a subset of AGU instructions that perform atomic read-modify-write operations on registers or memory locations. They execute in the BMU. All BMU instructions are listed in Table A-12. The architecture is described in [Section 2.3.6, “Bit Mask Instructions,”](#) on page 2-49. Although the BMU instructions are a subset of AGU instructions, they are presented in this Appendix with the other types for greater visibility, and to make them easier to find.

PREFIX Instructions - support conditional execution of other instructions and NOP insertion for time and space padding. They have unique properties since their binary form is a prefix encoding. They are decoded by the dispatcher, but are not dispatched to an execution unit. All PREFIX instructions are listed in Table A-16.

A.1.6.1 Instruction Sub-types

The instruction types can be further divided into sub-types as follows:

DALU Instruction Sub-types

- Data arithmetic (including multiply-accumulate) instructions are listed in Table A-7 and described in [Section 2.2.1.2, “Multiply-Accumulate \(MAC\) Unit,”](#) on page 2-10.
- Logical (including bit-field) instructions are listed in Table A-8 and described in [Section 2.2.1.3, “Bit-Field Unit \(BFU\),”](#) on page 2-12.

AGU Instruction Sub-types

- Address arithmetic instructions (AAU) are listed in Table A-9 and described in [Section 2.3.1, “AGU Architecture,”](#) on page 2-31.
- Move instructions are listed in Table A-10 and described in [Section 2.3.7, “Move Instructions,”](#) on page 2-51.
- Stack Support instructions are listed in Table A-11 and described in [Section 5.5, “Stack Support,”](#) on page 5-32.
- Bit-Mask (BMU) instructions are listed in Table A-12 and described in [Section 2.3.6, “Bit Mask Instructions,”](#) on page 2-49.
- Non-loop change-of-flow (non-loop COF) instructions are listed in Table A-13 and described in [Section 5.3.2, “Change-Of-Flow Instruction Timing,”](#) on page 5-17.
- Loop control instructions are listed in Table A-14 and described in [Section 5.4.6, “Loop Control Instructions,”](#) on page 5-29.
 - Loop change-of-flow instructions are also listed in Table A-14 and are described in [Section 5.3.2, “Change-Of-Flow Instruction Timing,”](#) on page 5-17.
- Program control instructions are listed in Table A-15 and described in [Section 5.7.1, “Processing State Change Instructions,”](#) on page 5-41.

Table A-7. DALU Arithmetic Instructions (MAC)

Instruction	Description
ABS	Absolute value
ADC	Add long with carry
ADD	Add
ADD2	Add two words
ADDNC.W	Add without changing the carry bit in the SR
ADR	Add and round
ASL	Arithmetic shift left by one bit
ASR	Arithmetic shift right by one bit
CLR	Clear
CMPEQ	Compare for equal
CMPGT	Compare for greater than
CMPHI	Compare for higher (unsigned)
DECEQ	Decrement a data register and set T if zero
DECGE	Decrement a data register and set T if greater than or equal to zero
DIV	Divide iteration
DMACSS	Multiply signed by signed and accumulate with data register right shifted by word size
DMACSU	Multiply signed by unsigned and accumulate with data register right shifted by word size
IADDNC.W	Add integers, 40-bit, non-saturating, immediate, no-carry update
IMAC	Multiply-accumulate integers
IMACLHUU	Multiply-accumulate unsigned integers; first source from low portion, second from high portion
IMACUS	Multiply-accumulate unsigned integer and signed integer
IMPY	Multiply signed integers in data registers
IMPY.W	Multiply signed immediate and signed integer in data register
IMPYHLUU	Multiply unsigned integer and unsigned integer; first source from high portion, second from low portion
IMPYSU	Multiply signed integer and unsigned integer
IMPYUU	Multiply unsigned integer and unsigned integer
INC	Increment a data register (as integer data)
INC.F	Increment a data register (as fractional data)
MAC	Multiply-accumulate signed fractions
MACR	Multiply-accumulate signed fractions and round
MACSU	Multiply-accumulate signed fraction and unsigned fraction
MACUS	Multiply-accumulate unsigned fraction and signed fraction
MACUU	Multiply-accumulate unsigned fraction and unsigned fraction
MAX	Transfer maximum signed value
MAX2	Transfer two 16-bit maximum signed values
MAX2VIT	Special MAX2 version for Viterbi kernel
MAXM	Transfer maximum magnitude value
MIN	Transfer minimum signed value
MPY	Multiply signed fractions

Table A-7. DALU Arithmetic Instructions (MAC) (Continued)

Instruction	Description
MPYR	Multiply signed fractions and round
MPYSU	Multiply signed fraction and unsigned fraction
MPYUS	Multiply unsigned fraction and signed fraction
MPYUU	Multiply unsigned fraction and unsigned fraction
NEG	Negate
RND	Round
SAT.F	Saturate fractional value in data register to fit in high portion
SAT.L	Saturate value in data register to fit in 32 bits
SBC	Subtract long with carry
SBR	Subtract and round
SUB	Subtract
SUB2	Subtract two words
SUBL	Shift left and subtract
SUBNC.W	Subtract without changing the carry bit in the status register
TFR	Transfer data register to a data register
TFRF	Conditional data register transfer, if the T bit is clear
TFRT	Conditional data register transfer, if the T bit is set
TSTEQ	Test for equal to zero
TSTGE	Test for greater than or equal to zero
TSTGT	Test for greater than zero

Table A-8. DALU Logical Instructions (BFU)

Instruction	Description
AND	Logical AND
ASLL	Multi-bit arithmetic shift left
ASLW	Word arithmetic shift left (16 bit shift)
ASRR	Multi-bit arithmetic shift right
ASRW	Word arithmetic shift right (16 bit shift)
CLB	Count leading bits (ones or zeros)
EOR	Logical exclusive OR
EXTRACT	Extract signed bit field
EXTRACTU	Extract unsigned bit field
INSERT	Insert bit field
LSLL	Multi-bit logical shift left
LSR	Logical shift right by one bit
LSRR	Multi-bit logical shift right
LSRW	Word logical shift right (16-bit shift)
NOT	One's complement (inversion)
OR	Logical inclusive OR

Table A-8. DALU Logical Instructions (BFU) (Continued)

Instruction	Description
ROL	Rotate one bit left through the carry bit
ROR	Rotate one bit right through the carry bit
SXT.B	Sign extend byte
SXT.L	Sign extend long
SXT.W	Sign extend word
ZXT.B	Zero extend byte
ZXT.L	Zero extend long
ZXT.W	Zero extend word

Table A-9. AGU Arithmetic Instructions

Instruction	Description
ADDA	Add (affected by the modifier mode)
ADDL1A	Add with 1-bit left shift of source operand (affected by the modifier mode)
ADDL2A	Add with 2-bit left shift of source operand (affected by the modifier mode)
ASL2A	Arithmetic shift left by 2 bits (32-bit)
ASLA	Arithmetic shift left (32-bit)
ASRA	Arithmetic shift right (32-bit)
CMPEQA	Compare for equal
CMPGTA	Compare for greater than
CMPHIA	Compare for higher (unsigned)
DECA	Decrement register
DECEQA	Decrement and set T if zero
DECGEA	Decrement and set T if equal or greater than zero
INCA	Increment register
LSRA	Logical shift right (32-bit)
SUBA	Subtract (affected by the modifier mode)
SXTA.B	Sign extend byte
SXTA.W	Sign extend word
TFRA	Register transfer
TFRA (OSP)	Move the "other" stack pointer to/from a register, inversely defined by the exception mode
TSTEQA.L	Test for equal
TSTEQA.W	Test for equal on lower 16 bits
TSTGEA	Test for greater than or equal
TSTGTA	Test for greater than
ZXTA.B	Zero extend byte
ZXTA.W	Zero extend word

Table A-10. AGU Move Instructions

Instruction	Description
MOVE.2F	Move two fractional words from memory to a register pair
MOVE.2L	Move two longs to/from a register pair
MOVE.2W	Move two integer words to/from memory and a register pair
MOVE.4F	Move four fractional words from memory to a register quad
MOVE.4W	Move four integer words to/from memory and a register quad
MOVE.B	Move byte to/from memory
MOVE.F	Move fractional word to/from memory
MOVE.L	Move long to/from memory
MOVE.W	Move integer word to/from memory, or immediate to register or memory
MOVEc	Move address register to address register, depending on T bit of SR
MOVES.2F	Move two fractional words to memory with scaling and limiting enabled
MOVES.4F	Move four fractional words to memory with scaling and limiting enabled
MOVES.F	Move fractional word to memory with scaling and limiting enabled
MOVES.L	Move long to memory with scaling and limiting enabled
MOVEU.B	Move unsigned byte from memory
MOVEU.L	Move unsigned long from immediate
VSL.2F	Viterbi shift left: special move for Viterbi kernel
VSL.2W	Viterbi shift left: special move for Viterbi kernel
VSL.4F	Viterbi shift left: special move for Viterbi kernel
VSL.4W	Viterbi shift left: special move for Viterbi kernel

Table A-11. AGU Stack Support Instructions

Instruction	Description
POP	Pop a register from the software stack
POPn	Pop a register from the software stack using the normal stack pointer
PUSH	Push a register onto the software stack
PUSHn	Push a register onto the software stack using the normal stack pointer

Table A-12. AGU Bit-Mask Instructions (BMU)

Instruction	Description
AND	Logical AND on a 16-bit operand
BMCHG	Bit-mask change a 16-bit operand
BMCHG.W	Bit-mask change a 16-bit operand in memory
BMCLR	Bit-mask clear a 16-bit operand
BMCLR.W	Bit-mask clear a 16-bit operand in memory
BMSET	Bit-mask set a 16-bit operand
BMSET.W	Bit-mask set a 16-bit operand in memory
BMTSET	Bit mask test and set a 16-bit operand
BMTSET.W	Bit mask test and set a 16-bit operand in memory
BMTSTC	Bit mask test if clear Sets the T-bit, if every bit position that has the value 1 in the mask is 0 in an operand.
BMTSTC.W	Bit mask test if clear in memory Sets the T-bit, if every bit position that has the value 1 in the mask is 0 in an operand.
BMTSTS	Bit mask test if set Sets the T-bit, if every bit position that has the value 1 in the mask is 1 in an operand.
BMTSTS.W	Bit mask test if set in memory Sets the T-bit, if every bit position that has the value 1 in the mask is 1 in an operand.
EOR	Logical Exclusive OR on a 16-bit operand
NOT	Binary inversion of a 16-bit operand
OR	Logical OR on a 16-bit operand

Table A-13. AGU Non-Loop Change-of-Flow Instructions

Instruction	Description
BF	Branch if false
BFD	Branch if false (delayed)
BRA	Branch
BRAD	Branch (delayed)
BSR	Branch to subroutine
BSRD	Branch to subroutine (delayed)
BT	Branch if true
BTD	Branch if true (delayed)
JF	Jump if false
JFD	Jump if false (delayed)
JMP	Jump
JMPD	Jump (delayed)
JSR	Jump to subroutine
JSRD	Jump to subroutine (delayed)
JT	Jump if true
JTD	Jump if true (delayed)
RTE	Return from exception

Table A-13. AGU Non-Loop Change-of-Flow Instructions

Instruction	Description
RTED	Return from exception (delayed)
RTS	Return from subroutine
RTSD	Return from subroutine (delayed)
RTSTK	Force restore PC from the stack, updating SP
RTSTKD	Force restore PC from the stack, updating SP (delayed)
TRAP	Execute a precise software exception

Table A-14. AGU Loop Control (Including Loop COF) Instructions

Instruction	Description
BREAK	Terminate the loop and branch to an address
CONT	Jump to the start of the loop to start the next iteration
CONTD	Jump to the start of the loop to start the next iteration (delayed)
DOENn	Do enable - set loop counter n and enable loop n as a long loop
DOENSHn	Do enable short - set loop counter n and enable loop n as a short loop
DOSETUPn	Setup loop start address n
SKIPLS	Test the active LC and skip the loop if LCn is equal or smaller than zero

Table A-15. AGU Program Control Instructions

Instruction	Description
DEBUG	Enter debug mode
DEBUGEV	Signal debug event
DI	Disable interrupts (sets the DI bit in the status register)
EI	Enable interrupts (clears the DI bit in the status register)
ILLEGAL	Trigger an imprecise illegal instruction exception
MARK	Push the PC into the trace buffer
STOP	Stop processing (lowest power stand-by)
WAIT	Wait for interrupt (low power stand-by)

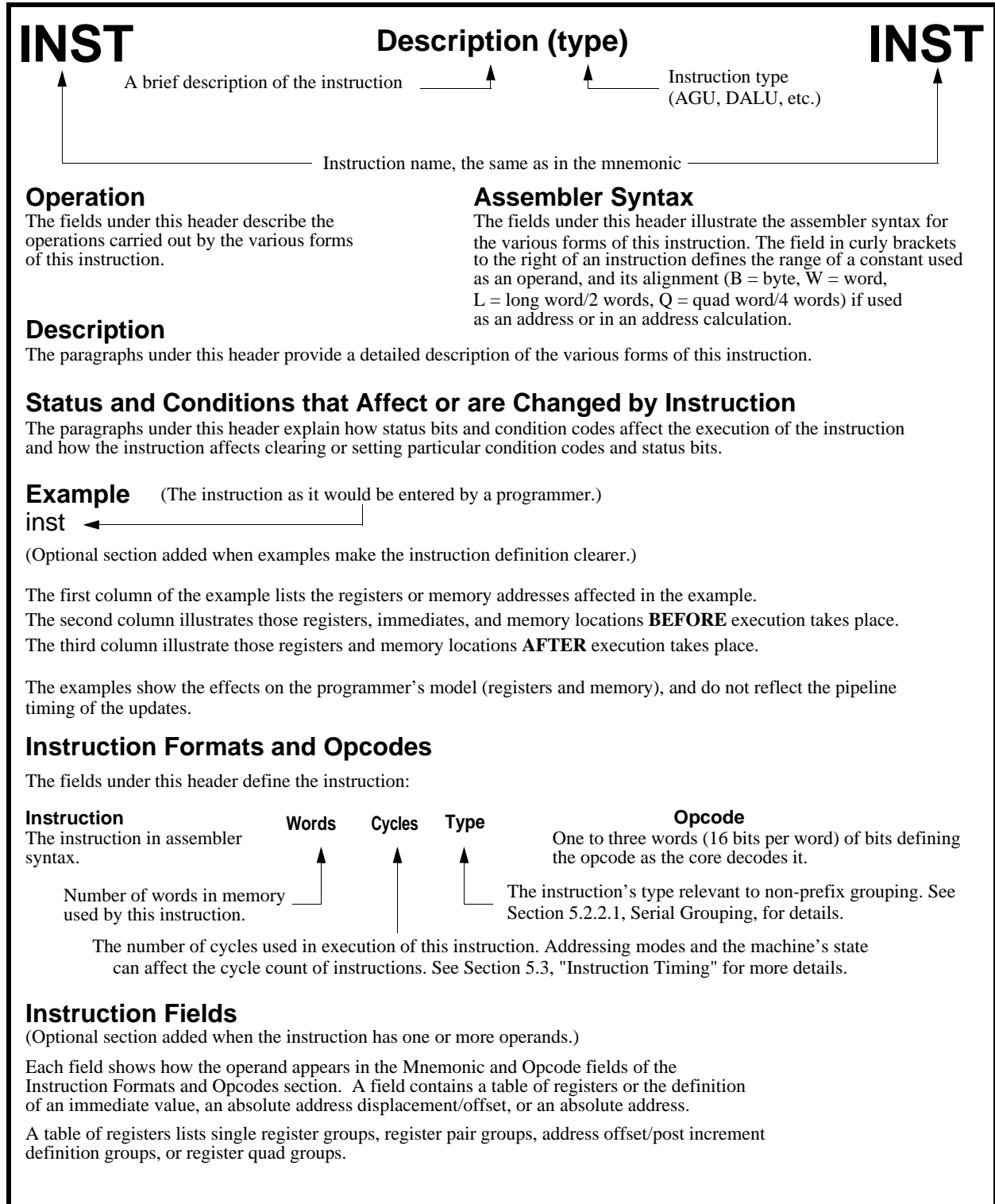
Table A-16. Prefix Instructions

Instruction	Description
IFA	Execute current execution set or subgroup unconditionally
IFF	Execute current execution set or subgroup if the T bit is clear
IFT	Execute current execution set or subgroup if the T bit is set
NOP	No operation

...2 Instructions

The following pages list all of the SC140 instructions and provide specific details about each instruction's operation and encoding.

A.2.1 Instruction Definition Layout



Operation

$|Dn| \rightarrow Dn$

Assembler Syntax

ABS Dn

Description

ABS Dn

Replaces the value in a data register (Dn) with its absolute value.

Status and Conditions that Affect Instruction

Register Address	Bit Name	Description
SR[2]	SM	If set, selects 32-bit arithmetic saturation mode.
SR[5:4]	S[1:0]	Scaling mode bits determine which bits in the result are used in the Ln bit calculation.

Status and Conditions Changed by Instruction

Register Address	Bit Name	Description
EMR[2]	DOVF	Set if the result cannot be represented in 40 bits, or if the result saturates to 32 bits in arithmetic saturation mode.
Ln	L	If not in arithmetic saturation mode (SR [SM] = 0), calculates and updates the Ln bit in the destination register. If in arithmetic saturation mode (SR [SM] = 1), clears the Ln bit in the destination register.

Example

abs d0

Register/Memory Address	Before	After
SR	\$00E0 0000	
L0:D0	\$0:FF FFFF FFF6	\$0:00 0000 000A
EMR		\$0000 0000

\$FFF6 = -10, \$000A = 10



Instruction Formats and Opcodes

Instruction	Words	Cycles	Type	Opcode																
ABS Dn	1	1	1	<div style="display: flex; justify-content: space-between; align-items: center;"> 15 8 7 0 </div> <table border="1" style="width: 100%; text-align: center; border-collapse: collapse;"> <tr> <td style="width: 12.5%;">0</td> <td style="width: 12.5%;">*</td> <td style="width: 12.5%;">1</td> <td style="width: 12.5%;">0</td> <td style="width: 12.5%;">0</td> <td style="width: 12.5%;">1</td> <td style="width: 12.5%;">F</td> <td style="width: 12.5%;">F</td> <td style="width: 12.5%;">F</td> <td style="width: 12.5%;">1</td> <td style="width: 12.5%;">1</td> <td style="width: 12.5%;">0</td> <td style="width: 12.5%;">0</td> <td style="width: 12.5%;">1</td> <td style="width: 12.5%;">1</td> <td style="width: 12.5%;">0</td> </tr> </table>	0	*	1	0	0	1	F	F	F	1	1	0	0	1	1	0
0	*	1	0	0	1	F	F	F	1	1	0	0	1	1	0					

Note: ** indicates serial grouping encoding.

Instruction Fields

Dn	FFF	Single Source/Destination Data Register						
	000	D0	010	D2	100	D4	110	D6
	001	D1	011	D3	101	D5	111	D7

Note: This instruction can specify D8-D15 as operands by using a prefix.

Operation

$Dc + Dd + C \rightarrow Dd$

Assembler Syntax

ADC Dc, Dd

Description

ADC Dc,Dd

Adds two source data registers (Dc, Dd) plus the carry bit and stores the result in the second data register (Dd). This instruction can be used in multiple precision addition as illustrated in the example, which is a 64-bit addition.

Note: The carry bit is set correctly for multiple precision arithmetic using long word operands if the extension of the destination data register is the sign-extension of bit 31.

Status and Conditions that Affect Instruction

Register Address	Bit Name	Description
SR[0]	C	Added as a carry bit to the LSB.
SR[5:4]	S[1:0]	The scaling mode bits determine which bits in the result are used in the Ln bit calculation.

Status and Conditions Changed by Instruction

Register Address	Bit Name	Description
EMR[2]	DOVF	Set if the result cannot be represented in 40 bits.
Ln	L	Calculates and updates the Ln bit in the destination register.
SR[0]	C	Calculates and updates the C bit in the status register.

Example

```
add d0,d1,d1 ;sets the carry bit
adc d4,d5 ;add with the carry bit
```

Register/Memory Address	Before	After
D0	\$FF 8000 0008	
L1:D1	\$0:\$FF 8000 0005	\$0:\$FF 0000 000D
SR	\$00E4 0000	\$00E4 0001
D4	\$00 0000 0005	
L5:D5	\$0:\$00 0000 0001	\$0:\$00 0000 0007
SR	\$00E4 00001	\$00E0 0000

Register/Memory Address Before

EMR

After

\$0000 0000

Instruction Formats and Opcodes

Instruction	Words	Cycles	Type	Opcode
ADC Dc, Dd	1	1	1	<div style="display: flex; justify-content: space-between; align-items: center;"> 15 8 7 0 </div> <div style="display: flex; justify-content: space-between; align-items: center;"> 0 * 1 0 1 1 e e 0 1 1 1 1 0 1 0 </div>

Note: ** indicates serial grouping encoding.

Instruction Fields

Dc, Dd	ee	Data Register Pairs					
00	D0, D1	01	D2, D3	10	D4, D5	11	D6, D7

Note: This instruction can specify D8-D15 as operands by using a prefix.

Operation

$\#u5 + Dn \rightarrow Dn$

$Da + Db \rightarrow Dn$

Assembler Syntax

`ADD #u5,Dn {0 ≤ u5 < 32}`

`ADD Da,Db,Dn`

Description

These operations add two source operands and store the result in a destination data register (Dn).

ADD #u5,Dn

The five bits of the unsigned immediate are right-aligned and the upper bits are zero-extended to form a 40-bit source operand. That operand is then added to a data register (Dn) and the result stored in the destination data register (Dn).

ADD Da,Db,Dn

Adds two source data registers (Da and Db) and stores the result in a destination data register (Dn).

Status and Conditions that Affect Instruction

Register Address	Bit Name	Description
SR[2]	SM	If set, selects 32-bit arithmetic saturation mode.
SR[5:4]	S[1:0]	Scaling mode bits determine which bits in the result are used in the Ln bit calculation.

Status and Conditions Changed by Instruction

Register Address	Bit Name	Description
SR[0]	C	Calculates and updates the C bit in the status register.
EMR[2]	DOVF	Set if the result cannot be represented in 40 bits, or if the result saturates to 32 bits in arithmetic saturation mode.
Ln	L	If not in arithmetic saturation mode (SR [SM] = 0), calculates and updates the Ln bit in the destination register. If in arithmetic saturation mode (SR [SM] = 1), clears the Ln bit in the destination register.

Example 1

```
add d0,d1,d2
```

Register/Memory Address	Before	After
SR	\$00E0 0000	
D0	\$00 0000 0005	
D1	\$00 0000 0002	

Register/Memory Address	Before	After
L2:D2		\$0:\$00 0000 0007
EMR		\$0000 0000

Example 2

add d1,d0,d2

Register/Memory Address	Before	After
SR	\$00E0 0000	
D1	\$00 72E3 8F2A	
D0	\$00 7216 EE3C	
L2:D2		\$1:\$00 E4FA 7D66
EMR		\$0000 0000

The L2 bit is set from the 32-bit overflow. Note that the extension bits are in use in the sum, bit 32 =0, bit 31 = 1.

Instruction Formats and Opcodes

Instruction	Words	Cycles	Type	Opcode								
ADD #u5,Dn	1	1	1	<table border="1"> <tr> <td>15</td> <td>8</td> <td>7</td> <td>0</td> </tr> <tr> <td>0 * 1 1 1 0 F F</td> <td>F 1 0</td> <td>i i i i</td> <td>i</td> </tr> </table>	15	8	7	0	0 * 1 1 1 0 F F	F 1 0	i i i i	i
15	8	7	0									
0 * 1 1 1 0 F F	F 1 0	i i i i	i									
ADD Da,Db,Dn	1	1	1	<table border="1"> <tr> <td>15</td> <td>8</td> <td>7</td> <td>0</td> </tr> <tr> <td>0 * 1 0 1 1 F F</td> <td>F 1 0</td> <td>J J J J</td> <td>J</td> </tr> </table>	15	8	7	0	0 * 1 0 1 1 F F	F 1 0	J J J J	J
15	8	7	0									
0 * 1 0 1 1 F F	F 1 0	J J J J	J									
ADD Da, Da, Dn	1	1	1	<table border="1"> <tr> <td>15</td> <td>8</td> <td>7</td> <td>0</td> </tr> <tr> <td>0 * 1 0 0 0 F F</td> <td>F 1 1 0 0 0</td> <td>j j</td> <td></td> </tr> </table>	15	8	7	0	0 * 1 0 0 0 F F	F 1 1 0 0 0	j j	
15	8	7	0									
0 * 1 0 0 0 F F	F 1 1 0 0 0	j j										

Note: ** indicates serial grouping encoding.



ADD2

Add Two 16-Bit Values (DALU)

ADD2

Operation

Da.H + Dn.H → Dn.H
Da.L + Dn.L → Dn.L

Assembler Syntax

ADD2 Da, Dn

Description

ADD2 Da,Dn

Performs a 32-bit addition of source registers Da and Dn with carry disabled between bits 15 and 16, so that the high and low words of each register are added separately. The result is stored back in Dn. The extension byte of the result is undefined.

Status and Conditions that Affect Instruction

None.

Status and Conditions Changed by Instruction

Register Address	Bit Name	Description
Ln	L	Clears the Ln bit in the destination register.

Example 1

```
add2 d0,d1
```

Register/Memory Address	Before	After
D0	\$00 1100 1100	
L1:D1	\$0:\$00 2200 3300	\$0:\$00 3300 4400

Example 2

```
add2 d0,d1
```

Register/Memory Address	Before	After
D0	\$00 1101 F011	
L1:D1	\$0:\$00 0020 2002	\$0:\$00 1121 1013

Instruction Formats and Opcodes

Instruction	Words	Cycles	Type	Opcode																
ADD2 Da, Dn	1	1	2	<div style="display: flex; justify-content: space-between; align-items: center;"> 15 8 7 0 </div> <table border="1" style="width: 100%; text-align: center; border-collapse: collapse;"> <tr> <td style="width: 25%;">1</td> <td style="width: 25%;">1</td> <td style="width: 25%;">0</td> <td style="width: 25%;">1</td> <td style="width: 25%;">0</td> <td style="width: 25%;">0</td> <td style="width: 25%;">F</td> <td style="width: 25%;">F</td> <td style="width: 25%;">F</td> <td style="width: 25%;">1</td> <td style="width: 25%;">0</td> <td style="width: 25%;">0</td> <td style="width: 25%;">0</td> <td style="width: 25%;">J</td> <td style="width: 25%;">J</td> <td style="width: 25%;">J</td> </tr> </table>	1	1	0	1	0	0	F	F	F	1	0	0	0	J	J	J
1	1	0	1	0	0	F	F	F	1	0	0	0	J	J	J					

Instruction Fields

Dn **FFF** **Single Source/Destination Data Register**

000	D0	010	D2	100	D4	110	D6
001	D1	011	D3	101	D5	111	D7

Note: This instruction can specify D8-D15 as operands by using a prefix.

Da **JJJ** **Single Source Data Register**

000	D0	010	D2	100	D4	110	D6
001	D1	011	D3	101	D5	111	D7

Note: This instruction can specify D8-D15 as operands by using a prefix.



ADDA

Add (AGU)

ADDA

Operation

$\#u5 + Rx \rightarrow Rx$

$\#s16 + rx \rightarrow Rn$

$rx + Rx \rightarrow Rx$

Assembler Syntax

ADDA #u5,Rx $\{0 \leq u5 < 32\}$

ADDA #s16,rx,Rn $\{-2^{15} \leq s16 < 2^{15}\}$

ADDA rx,Rx

Description

These operations add an immediate signed 16-bit integer to the contents of a source AGU register and store the result in a destination address register. If the second source operand (rx) uses R0-R7, the operation is affected by the modifier mode selected in the modifier control register (MCTL).

ADDA #u5,Rx

Adds an immediate unsigned 5-bit integer to a source AGU register, Rx, (address or offset register, program counter, or active stack pointer) and stores the result in the destination register (Rx). The five bits of the unsigned integer are right-aligned and the upper bits are zero-extended to form a 32-bit source operand. For R0-R7, the operation is affected by the modifier mode selected in MCTL. If the stack pointer is the destination operand, then the immediate value must be a multiple of eight as its three LSBs are forced to zero.

ADDA #s16,rx,Rn

Adds an immediate signed 16-bit integer and the contents of a source AGU register (rx) and stores the result in a destination address register (Rn). The 16 bits of the signed integer are right-aligned and the upper bits are sign-extended to form a 32-bit operand. If the second source operand (rx) uses R0-R7, the operation is affected by the modifier mode selected in MCTL.

ADDA rx,Rx

Adds the contents of two source AGU registers (rx, Rx) and stores the result in the destination (second source) register (Rx). If the second source operand (Rx) uses R0-R7, the operation is affected by the modifier mode selected in MCTL. If the stack pointer is the destination operand, then the value in rx must be a multiple of eight as its three LSBs are forced to zero.

Status and Conditions that Affect Instruction

Register Address	Bit Name	Description
SR[18]	EXP	Determines which stack pointer is used when the stack pointer is an operand. Otherwise, the instruction is not affected by SR.
MCTL[31:0]	AM3–AM0	Address modification bits when updating R0–R7. Otherwise, the instruction is not affected by MCTL.

Status and Conditions Changed by Instruction

None.

Example 1

```
adda r0,r1
```

Register/Memory Address	Before	After
MCTL	\$0000 0000	
R0	\$0000 1100	
R1	\$0000 2200	\$0000 3300

Example 2

```
move.l #$8,mctl ;assigns m0 to r0, modulo arithmetic
move.l #$10,m0 ;puts modulo 16 in m0
move.w #$c,r0 ;initializes 12 to r0
nop
adda #$8,r0,r1 ; 8 + 12 = 20 or 4 modulo 16
```

Register/Memory Address	Before	After
MCTL	\$00000008	
R0	\$0000000c	
R1	\$00000000	\$00000004

Instruction Formats and Opcodes

Instruction	Words	Cycles	Type	Opcode												
ADDA #u5,Rx	1	1	2	<table border="1"> <tr> <td>15</td> <td>8</td> <td>7</td> <td>0</td> </tr> <tr> <td>1 1 1 0 R R R R</td> <td>0 1 0 i i i i i</td> <td></td> <td></td> </tr> </table>	15	8	7	0	1 1 1 0 R R R R	0 1 0 i i i i i						
15	8	7	0													
1 1 1 0 R R R R	0 1 0 i i i i i															
ADDA #s16,rx,Rn	2	1	4	<table border="1"> <tr> <td>15</td> <td>8</td> <td>7</td> <td>0</td> </tr> <tr> <td>0 0 1 0 r r r r</td> <td>i i i 0 1 R R R</td> <td></td> <td></td> </tr> <tr> <td>1 0 0 i i i i i</td> <td>i i i i i i i i</td> <td></td> <td></td> </tr> </table>	15	8	7	0	0 0 1 0 r r r r	i i i 0 1 R R R			1 0 0 i i i i i	i i i i i i i i		
15	8	7	0													
0 0 1 0 r r r r	i i i 0 1 R R R															
1 0 0 i i i i i	i i i i i i i i															
ADDA rx,Rx	1	1	2	<table border="1"> <tr> <td>15</td> <td>8</td> <td>7</td> <td>0</td> </tr> <tr> <td>1 1 1 0 R R R R</td> <td>0 0 0 1 r r r r</td> <td></td> <td></td> </tr> </table>	15	8	7	0	1 1 1 0 R R R R	0 0 0 1 r r r r						
15	8	7	0													
1 1 1 0 R R R R	0 0 0 1 r r r r															

Instruction Fields

Rn	RRR		Address Register				
000	R0	010	R2	100	R4	110	R6
001	R1	011	R3	101	R5	111	R7

Note: This instruction can specify R8-R15 as operands by using a high register prefix.

rrrr **AGU Source Register**

0000	N0	0100	—	1000	R0	1100	R4
0001	N1	0101	—	1001	R1	1101	R5
0010	N2	0110	PC	1010	R2	1110	R6
0011	N3	0111	SP	1011	R3	1111	R7

Note: This instruction can specify R8-R15 as operands by using a high register prefix.

Rx **RRRR** **AGU Source/Destination Register**

0000	N0	0100	—	1000	R0	1100	R4
0001	N1	0101	—	1001	R1	1101	R5
0010	N2	0110	—	1010	R2	1110	R6
0011	N3	0111	SP	1011	R3	1111	R7

Note: This instruction can specify R8-R15 as operands by using a high register prefix.

#u5 iiii 5-bit unsigned immediate data

#s16 iiiiiiiiiiiiiiii 16-bit signed immediate data



ADDL1A Add With One-Bit Arithmetic Shift Left of Source Operand (AGU) ADDL1A

Operation

$(rx \ll 1) + Rx \rightarrow Rx$

Assembler Syntax

ADDL1A rx, Rx

Description

ADDL1A rx, Rx

Performs a one-bit arithmetic shift left on the data from source AGU register (rx) and adds the result to a second source AGU register (Rx). The sum is stored back in Rx. For R0-R7 destinations, the operation is affected by the modifier mode selected in MCTL.

Status and Conditions that Affect Instruction

Register Address	Bit Name	Description
SR[18]	EXP	Determines which stack pointer is used when the stack pointer is an operand. Otherwise, the instruction is not affected by SR.
MCTL[31:0]	AM3-AM0	Address modification bits when updating R0-R7. Otherwise, the instruction is not affected by MCTL.

Example

addl1a r0, r1

Register/Memory Address	Before	After
MCTL	\$0000 0000	
R0	\$0000 0055	
R1	\$0000 0011	\$0000 00BB

In binary:

R0	01010101
R0 shifted left	10101010
R1	00010001
Sum	10111011



Instruction Formats and Opcodes

Instruction	Words	Cycles	Type	Opcode		
ADDL1A rx,Rx	1	1	2	<div style="display: flex; justify-content: space-between; align-items: center;"> 15 8 7 0 </div> <table border="1" style="width: 100%; text-align: center; border-collapse: collapse;"> <tr> <td style="width: 40%;">1 1 1 0 R R R R</td> <td style="width: 60%;">0 0 0 0 r r r r</td> </tr> </table>	1 1 1 0 R R R R	0 0 0 0 r r r r
1 1 1 0 R R R R	0 0 0 0 r r r r					

Instruction Fields

rx	rrrr	AGU Source Register					
0000	N0	0100	—	1000	R0	1100	R4
0001	N1	0101	—	1001	R1	1101	R5
0010	N2	0110	PC	1010	R2	1110	R6
0011	N3	0111	SP	1011	R3	1111	R7

Note: This instruction can specify R8-R15 as operands by using a high register prefix.

Rx	RRRR	AGU Source/Destination Register					
0000	N0	0100	—	1000	R0	1100	R4
0001	N1	0101	—	1001	R1	1101	R5
0010	N2	0110	—	1010	R2	1110	R6
0011	N3	0111	SP	1011	R3	1111	R7

Note: This instruction can specify R8-R15 as operands by using a high register prefix.



ADDL2A Add With Two-Bit Arithmetic Shift Left of Source Operand (AGU) ADDL2A

Operation

$(rx \ll 2) + Rx \rightarrow Rx$

Assembler Syntax

ADDL2A rx, Rx

Description

ADDL2A rx, Rx

Performs a two-bit arithmetic shift left on the data from AGU source register (rx), adds the result to another AGU source register (Rx), and stores the sum in the destination (second) register (Rx). For R0-R7 destinations, the operation is affected by the modifier mode selected in MCTL.

Status and Conditions that Affect Instruction

Register Address	Bit Name	Description
SR[18]	EXP	Determines which stack pointer is used when the stack pointer is an operand. Otherwise, the instruction is not affected by SR.
MCTL[31:0]	AM3–AM0	Address modification bits when updating R0–R7. Otherwise, the instruction is not affected by MCTL.

Status and Conditions Changed by Instruction

None.

Example

```
addl2a r0, r1
```

Register/Memory Address	Before	After
MCTL	\$0000 0000	
R0	\$0000 0055	
R1	\$0000 0011	\$0000 0165

In binary:

R0	01010101
R0 shifted left two	101010101
R1	00010001
Sum	101100101



Instruction Formats and Opcodes

Instruction	Words	Cycles	Type	Opcode
ADDL2A rx,Rx	1	1	2	<div style="display: flex; justify-content: space-between; align-items: center;"> 15 8 7 0 </div> <div style="border: 1px solid black; padding: 2px; display: flex; justify-content: space-between; align-items: center;"> 1 1 1 0 R R R R 0 0 1 0 r r r r </div>

Instruction Fields

rx	rrrr	AGU Source Register					
0000	N0	0100	—	1000	R0	1100	R4
0001	N1	0101	—	1001	R1	1101	R5
0010	N2	0110	PC	1010	R2	1110	R6
0011	N3	0111	SP	1011	R3	1111	R7

Note: This instruction can specify R8-R15 as operands by using a high register prefix.

Rx	RRRR	AGU Source/Destination Register					
0000	N0	0100	—	1000	R0	1100	R4
0001	N1	0101	—	1001	R1	1101	R5
0010	N2	0110	—	1010	R2	1110	R6
0011	N3	0111	SP	1011	R3	1111	R7

Note: This instruction can specify R8-R15 as operands by using a high register prefix.



ADDNC.W

Add Without Changing
the Carry Bit (DALU)

ADDNC.W

Operation

$\#s16 + Da \rightarrow Dn$

Assembler Syntax

`ADDNC.W #s16,Da,Dn` $\{-2^{15} \leq s16 < 2^{15}\}$

Description

`ADDNC.W #s16,Da,Dn`

Sign-extends the 16-bit immediate value to 40 bits and adds it to the source data register Da. The sum is stored in destination register Dn. The carry bit is not affected by this instruction.

Status and Conditions that Affect Instruction

Register Address	Bit Name	Description
SR[2]	SM	If set, selects 32-bit arithmetic saturation mode.
SR[5:4]	S[1:0]	The scaling mode bits determine which bits in the result are used in the Ln bit calculation.

Status and Conditions Changed By Instruction

Register Address	Bit Name	Description
EMR[3]	DOVF	Set if the result cannot be represented in 40 bits, or if the result saturates to 32 bits in arithmetic saturation mode.
Ln	L	If not in arithmetic saturation mode (SR [SM] = 0), calculates and updates the Ln bit in the destination register. If in arithmetic saturation mode (SR [SM] = 1), clears the Ln bit in the destination register.

Example

```
addnc.w #0xca3e,d1,d2
```

Register/Memory Address	Before	After
immediate operand	\$FF FFFF CA3E	
D1	\$FF FFFF CA3E	
L2:D2		\$0:\$FF FFFF 947C
SR	\$00E0 0000	\$00E0 0000
EMR		\$0000 0000

An add with a carry allowed would result in setting the carry bit as a result of an overflow from bit 39.



Instruction Formats and Opcodes

Instruction	Words	Cycles	Type	Opcode																																
ADDNC #s16, Da, Dn	2	1	4	<div style="display: flex; justify-content: space-between; width: 100%;"> 15 8 7 0 </div> <table border="1" style="width: 100%; text-align: center;"> <tr> <td>0</td><td>0</td><td>1</td><td>1</td><td>J</td><td>J</td><td>J</td><td>1</td> <td>i</td><td>i</td><td>i</td><td>0</td><td>0</td><td>F</td><td>F</td><td>F</td> </tr> <tr> <td>1</td><td>0</td><td>0</td><td>i</td><td>i</td><td>i</td><td>i</td><td>i</td> <td>i</td><td>i</td><td>i</td><td>i</td><td>i</td><td>i</td><td>i</td><td>i</td> </tr> </table>	0	0	1	1	J	J	J	1	i	i	i	0	0	F	F	F	1	0	0	i	i	i	i	i	i	i	i	i	i	i	i	i
0	0	1	1	J	J	J	1	i	i	i	0	0	F	F	F																					
1	0	0	i	i	i	i	i	i	i	i	i	i	i	i	i																					

Instruction Fields

Da	JJJ	Single Source Data Register					
000	D0	010	D2	100	D4	110	D6
001	D1	011	D3	101	D5	111	D7

Note: This instruction can specify D8-D15 as operands by using a prefix.

Dn	FFF	Single Source/Destination Data Register					
000	D0	010	D2	100	D4	110	D6
001	D1	011	D3	101	D5	111	D7

Note: This instruction can specify D8-D15 as operands by using a prefix.

#s16 iiiiiiiiiiiiii 16-bit signed immediate data

Operation

$\text{Rnd}(\text{Da} + \text{Dn}) \rightarrow \text{Dn}$

Assembler Syntax

ADR Da, Dn

Description

ADR Da,Dn

Adds one source data register (Da) to another (Dn) and rounds the sum. The result is stored in the destination data register (Dn). Rounding adjusts the LSB of the high part of the destination register according to the value of the low part of the register and then zeros the low part. The boundary between the high part and the low part changes with scaling. The two modes of the round function, Rnd(), are described on page A-359.

Status and Conditions that Affect Instruction

Register Address	Bit Name	Description
SR[2]	SM	If set, selects 32-bit arithmetic saturation mode.
SR[3]	RM	Rounding mode
SR[5:4]	S[1:0]	The scaling mode bits determine which bits in the result are used in the Ln bit calculation, and which bits are used in rounding.

Status and Conditions Changed by Instruction

Register Address	Bit Name	Description
EMR[2]	DOVF	Set if the result cannot be represented in 40 bits, or if the result saturates to 32 bits in arithmetic saturation mode.
Ln	L	If not in arithmetic saturation mode (SR [SM] = 0), calculates and updates the Ln bit in the destination register. If in arithmetic saturation mode (SR [SM] = 1), clears the Ln bit in the destination register.

Example

adr d3,d4

Register/Memory Address	Before	After
D3	\$00 0034 A216	
L4:D4	\$0:\$00 2000 0000	\$0:\$00 2035 0000
SR	\$00E0 0000	
EMR		\$0000 0000



Instruction Formats and Opcodes

Instruction	Words	Cycles	Type	Opcode																
ADR Da, Dn	1	1	1	<div style="display: flex; justify-content: space-between; align-items: center;"> 15 8 7 0 </div> <table border="1" style="width: 100%; text-align: center;"> <tr> <td style="width: 10%;">0</td> <td style="width: 10%;">*</td> <td style="width: 10%;">1</td> <td style="width: 10%;">1</td> <td style="width: 10%;">0</td> <td style="width: 10%;">0</td> <td style="width: 10%;">F</td> <td style="width: 10%;">F</td> <td style="width: 10%;">F</td> <td style="width: 10%;">1</td> <td style="width: 10%;">0</td> <td style="width: 10%;">0</td> <td style="width: 10%;">0</td> <td style="width: 10%;">J</td> <td style="width: 10%;">J</td> <td style="width: 10%;">J</td> </tr> </table>	0	*	1	1	0	0	F	F	F	1	0	0	0	J	J	J
0	*	1	1	0	0	F	F	F	1	0	0	0	J	J	J					

Note: ** indicates serial grouping encoding.

Instruction Fields

Da **JJJ** **Single Source Data Register**

000	D0	010	D2	100	D4	110	D6
001	D1	011	D3	101	D5	111	D7

Note: This instruction can specify D8-D15 as operands by using a prefix.

Dn **FFF** **Single Source/Destination Data Register**

000	D0	010	D2	100	D4	110	D6
001	D1	011	D3	101	D5	111	D7

Note: This instruction can specify D8-D15 as operands by using a prefix.

Operation

$\#0u16 \bullet Da \rightarrow Dn$

$\#u16\$0000 \bullet Da \rightarrow Dn$

$Da \bullet Dn \rightarrow Dn$

Assembler Syntax

`AND #0{u16},Da,Dn $\{0 \leq u16 < 2^{16}\}$`

`AND #{u16}$0000,Da,Dn $\{0 \leq u16 < 2^{16}\}$`

`AND Da,Dn`

Description

These operations perform a "logical and" between the two source operands, and store the result in the destination operand.

AND #0{u16},Da,Dn

The immediate unsigned word is zero-extended in bits [40:16] to form a 40-bit immediate operand. This operand is then ANDed with the contents of a source data register (Da), and the result stored in a destination data register (Dn). The HP and extension (bits [40:16]) of the destination register are cleared as a result of this instruction.

The { } are not part of the assembler syntax, they are used here for clarity. For example, given an immediate value of \$27A6, using D0 as the source data register, and using D1 as the destination data register, this instruction would be written as:

```
and #027a6,d0,d1
```

AND #{u16}\$0000,Da,Dn

A 40-bit operand is formed with zeros in bits [15:0], the immediate word in bits [31:16], and bit 31 copied to bits [39:32] (sign-extended). This operand is then ANDed with the contents of a source data register (Da), and the result stored in a destination data register (Dn). The LP of the destination register is cleared as a result of this instruction.

The { } are not part of the assembler syntax, they are used here for clarity. For example, given an immediate value of \$27A6, using D0 as the source data register, and using D1 as the destination data register, this instruction would be written as:

```
and #27a60000,d0,d1
```

AND Da,Dn

Performs a bitwise AND on the contents of two source data registers (Da, Dn) and stores the result in the destination (second) data register (Dn).

Status and Conditions that Affect Instruction

None.

Status and Conditions Changed by Instruction

Register Address	Bit Name	Description
Ln	L	Clears the Ln bit in the destination register.

Example 1

and d2,d1

Register/Memory Address	Before	After
D2	\$FF CE66 47F2	
L1:D1	\$0:\$FF D859 6705	\$0:\$FF C840 4700

Example 2

and #ff2e,d2,d1

Register/Memory Address	Before	After
immediate	\$00 0000 FF2E	
D2	\$00 27A6 98FB	
L1:D1		\$0:\$00 0000 982A

Example 3

and #ff2e0000,d2,d1

Register/Memory Address	Before	After
immediate	\$FF FF2E 0000	
D2	\$F0 27A6 98FB	
L1:D1		\$0:\$F0 2726 0000

Note: The value of the immediate ff2e0000 is extended to ffff2e0000 before the AND operation with D2.

Instruction Formats and Opcodes

Instruction	Words	Cycles	Type	Opcode												
AND #0{u16},Da,Dn	2	1	4	<table border="1"> <tr> <td>15</td> <td>8</td> <td>7</td> <td>0</td> </tr> <tr> <td>0 0 1 1 J J J 1</td> <td>i i i 1 1 F F F</td> <td></td> <td></td> </tr> <tr> <td>1 0 0 i i i i i</td> <td>i i i i i i i i</td> <td></td> <td></td> </tr> </table>	15	8	7	0	0 0 1 1 J J J 1	i i i 1 1 F F F			1 0 0 i i i i i	i i i i i i i i		
15	8	7	0													
0 0 1 1 J J J 1	i i i 1 1 F F F															
1 0 0 i i i i i	i i i i i i i i															
AND #{u16}\$0000,Da,Dn	2	1	4	<table border="1"> <tr> <td>15</td> <td>8</td> <td>7</td> <td>0</td> </tr> <tr> <td>0 0 1 1 J J J 1</td> <td>i i i 0 1 F F F</td> <td></td> <td></td> </tr> <tr> <td>1 0 0 i i i i i</td> <td>i i i i i i i i</td> <td></td> <td></td> </tr> </table>	15	8	7	0	0 0 1 1 J J J 1	i i i 0 1 F F F			1 0 0 i i i i i	i i i i i i i i		
15	8	7	0													
0 0 1 1 J J J 1	i i i 0 1 F F F															
1 0 0 i i i i i	i i i i i i i i															
AND Da,Dn	1	1	2	<table border="1"> <tr> <td>15</td> <td>8</td> <td>7</td> <td>0</td> </tr> <tr> <td>1 1 0 1 1 1 F F</td> <td>F 0 0 0 0 J J J</td> <td></td> <td></td> </tr> </table>	15	8	7	0	1 1 0 1 1 1 F F	F 0 0 0 0 J J J						
15	8	7	0													
1 1 0 1 1 1 F F	F 0 0 0 0 J J J															

Instruction Fields

Da **JJJ** **Single Source Data Register**

000	D0	010	D2	100	D4	110	D6
001	D1	011	D3	101	D5	111	D7

Note: This instruction can specify D8-D15 as operands by using a prefix.

Dn **FFF** **Single Source/Destination Data Register**

000	D0	010	D2	100	D4	110	D6
001	D1	011	D3	101	D5	111	D7

Note: This instruction can specify D8-D15 as operands by using a prefix.

#0(u16)	0000000000000000 iiiiiiiiiiiiiiii	16-bit unsigned immediate data in lower word, upper word zeroed
#{u16}\$0000	iiiiiiiiiiiiiiii 0000000000000000	16-bit unsigned immediate data in upper word, lower word zeroed
#u16	iiiiiiiiiiiiiiii	16-bit unsigned immediate data



AND

Bitwise AND with 16-Bit Immediate (BMU)

AND

Operation

#u16 • DR.L → DR.L

#u16 • DR.H → DR.H

Assembler Syntax

AND #u16,DR.L

AND #u16,DR.H

Description

AND #u16,DR.L

Performs a bitwise AND on an immediate unsigned word and the contents of the LP of a source data or address register (DR). Stores the result in the LP of the data or address register (DR). The HP of the register is unaffected.

Note: This instruction is assembler-mapped to BMCLR #~u16,DR.L where #~u16 is the bitwise complement of #u16.

AND #u16,DR.H

Performs a bitwise AND on an immediate unsigned word and the contents of the HP of a data or address register (DR). Stores the result in the HP of the data or address register (DR). The LP of the register is unaffected.

Note: This instruction is assembler-mapped to BMCLR #~u16,DR.H.

Status and Conditions that Affect Instruction

None.

Status and Conditions Changed by Instruction

Register Address	Bit Name	Description
Ln	L	Clears the Ln bit in the destination register.

Example

and #a70e,d1.h

Register/Memory Address	Before	After
immediate	\$A70E	
D1.H	\$57AF	\$070E

In binary, \$A70E	1010011100001110
\$57AF	0101011110101111
and = \$070E	0000011100001110

Instruction Formats and Opcodes

Instruction	Words	Cycles	Type	Opcode																																
AND #u16,DR.L	2	2	3	<div style="display: flex; justify-content: space-between; width: 100%;"> 15 8 7 0 </div> <table border="1" style="width: 100%; border-collapse: collapse;"> <tr> <td style="width: 8px;">0</td><td style="width: 8px;">0</td><td style="width: 8px;">0</td><td style="width: 8px;">0</td><td style="width: 8px;">1</td><td style="width: 8px;">0</td><td style="width: 8px;">0</td><td style="width: 8px;">0</td> <td style="width: 8px;">i</td><td style="width: 8px;">i</td><td style="width: 8px;">i</td><td style="width: 8px;">0</td><td style="width: 8px;">H</td><td style="width: 8px;">H</td><td style="width: 8px;">H</td><td style="width: 8px;">H</td> </tr> <tr> <td>1</td><td>0</td><td>1</td><td>i</td><td>i</td><td>i</td><td>i</td><td>i</td> <td>i</td><td>i</td><td>i</td><td>i</td><td>i</td><td>i</td><td>i</td><td>i</td> </tr> </table>	0	0	0	0	1	0	0	0	i	i	i	0	H	H	H	H	1	0	1	i	i	i	i	i	i	i	i	i	i	i	i	i
0	0	0	0	1	0	0	0	i	i	i	0	H	H	H	H																					
1	0	1	i	i	i	i	i	i	i	i	i	i	i	i	i																					
AND #u16,DR.H	2	2	3	<div style="display: flex; justify-content: space-between; width: 100%;"> 15 8 7 0 </div> <table border="1" style="width: 100%; border-collapse: collapse;"> <tr> <td style="width: 8px;">0</td><td style="width: 8px;">0</td><td style="width: 8px;">0</td><td style="width: 8px;">0</td><td style="width: 8px;">1</td><td style="width: 8px;">0</td><td style="width: 8px;">0</td><td style="width: 8px;">0</td> <td style="width: 8px;">i</td><td style="width: 8px;">i</td><td style="width: 8px;">i</td><td style="width: 8px;">1</td><td style="width: 8px;">H</td><td style="width: 8px;">H</td><td style="width: 8px;">H</td><td style="width: 8px;">H</td> </tr> <tr> <td>1</td><td>0</td><td>1</td><td>i</td><td>i</td><td>i</td><td>i</td><td>i</td> <td>i</td><td>i</td><td>i</td><td>i</td><td>i</td><td>i</td><td>i</td><td>i</td> </tr> </table>	0	0	0	0	1	0	0	0	i	i	i	1	H	H	H	H	1	0	1	i	i	i	i	i	i	i	i	i	i	i	i	i
0	0	0	0	1	0	0	0	i	i	i	1	H	H	H	H																					
1	0	1	i	i	i	i	i	i	i	i	i	i	i	i	i																					

Instruction Fields

DR	HHHH	Data/Address Register					
0000	D0	0100	D4	1000	R0	1100	R4
0001	D1	0101	D5	1001	R1	1101	R5
0010	D2	0110	D6	1010	R2	1110	R6
0011	D3	0111	D7	1011	R3	1111	R7

Note: This instruction can specify D8-D15 or R8-R15 as operands by using a a prefix prefix.

#u16 ~iiiiiiiiiiiiiii One's complement of 16-bit unsigned immediate data



AND.W Bitwise AND with 16-Bit Immediate (BMU) AND.W

Operation

$\#u16 \bullet (R) \rightarrow (R)$

$\#u16 \bullet (SP - u5) \rightarrow (SP - u5)$

$\#u16 \bullet (a16) \rightarrow (a16)$

$\#u16 \bullet (SP + s16) \rightarrow (SP + s16)$

Assembler Syntax

`AND.W #u16, (Rn) { $0 \leq u16 < 2^{16}$ }`

`AND.W #u16, (SP-u5) { $0 \leq u16 < 2^{16}$ } { $0 \leq u5 < 64, W$ }`

`AND.W #u16, (a16) { $0 \leq u16 < 2^{16}$ } { $0 \leq a16 < 2^{16}, W$ }`

`AND.W #u16, (SP+s16) { $0 \leq u16 < 2^{16}$ } { $-2^{15} \leq s16 < 2^{15}, W$ }`

Description

These operations read from memory, modify the retrieved value, and write the new value back to that memory address, resulting in two memory accesses. The absolute addresses, offsets, and address register values must be word-aligned.

AND.W #u16,(Rn)

Performs a bitwise AND on a 16-bit unsigned immediate value and the contents of a memory address, pointed to by the contents of an address register (Rn). Stores the result in the same memory address.

Note: This instruction is assembler-mapped to `BMCLR.W #~u16,(Rn)` where `#~u16` is the one's complement of `#u16`.

AND.W #u16,(SP-u5)

Performs a bitwise AND on a 16-bit unsigned immediate value and the contents of a memory address, pointed to by a 5-bit unsigned offset subtracted from SP. Stores the result in the same memory address. The address offset must be even.

Note: This instruction is assembler-mapped to `BMCLR.W #~u16,(SP-u5)`.

AND.W #u16,(a16)

Performs a bitwise AND on a 16-bit unsigned immediate value and the contents of a 16-bit absolute memory address. Stores the result in the same memory address.

Note: This instruction is assembler-mapped to `BMCLR.W #~u16, (a16)`.

AND.W #u16,(SP+s16)

Performs a bitwise AND on a 16-bit unsigned immediate value and the contents of a memory address, pointed to by a 15-bit signed offset added to SP. Stores the result in the same memory address.

Note: This instruction is assembler-mapped to `BMCLR.W #~u16,(SP+s16)`.

Status and Conditions that Affect Instruction

Register Address	Bit Name	Description
SR[18]	EXP	Determines which stack pointer is used when the stack pointer is an operand. Otherwise, the instruction is not affected by SR.

Status and Conditions Changed by Instruction

None.

Example

`and.w #$54a1, (r7)`

Register/Memory Address	Before	After
immediate	\$54A1	
R7	\$50	
(\$50)	\$15AF	\$14A1
In binary, \$54A1	0101010010100001	
\$15AF	0001010110101111	
and = \$14A1	0001010010100001	



Instruction Formats and Opcodes

Instruction	Words	Cycles	Type	Opcode																
AND.W #u16,(Rn)	2	2	3	<table border="1"> <tr> <td>15</td> <td>8</td> <td>7</td> <td>0</td> </tr> <tr> <td>0 0 0 1 0 0 0 0</td> <td>i i i 0 1 R R R</td> <td></td> <td></td> </tr> <tr> <td>1 0 1 i i i i i</td> <td>i i i i i i i i</td> <td></td> <td></td> </tr> </table>	15	8	7	0	0 0 0 1 0 0 0 0	i i i 0 1 R R R			1 0 1 i i i i i	i i i i i i i i						
15	8	7	0																	
0 0 0 1 0 0 0 0	i i i 0 1 R R R																			
1 0 1 i i i i i	i i i i i i i i																			
AND.W #u16,(SP-u5)	2	3	3	<table border="1"> <tr> <td>15</td> <td>8</td> <td>7</td> <td>0</td> </tr> <tr> <td>0 0 0 0 0 0 0 0</td> <td>i i i A A A A A</td> <td></td> <td></td> </tr> <tr> <td>1 0 1 i i i i i</td> <td>i i i i i i i i</td> <td></td> <td></td> </tr> </table>	15	8	7	0	0 0 0 0 0 0 0 0	i i i A A A A A			1 0 1 i i i i i	i i i i i i i i						
15	8	7	0																	
0 0 0 0 0 0 0 0	i i i A A A A A																			
1 0 1 i i i i i	i i i i i i i i																			
AND.W #u16,(a16)	3	2	3	<table border="1"> <tr> <td>15</td> <td>8</td> <td>7</td> <td>0</td> </tr> <tr> <td>0 0 1 1 1 0 0 0</td> <td>A A A i i 0 0 1</td> <td></td> <td></td> </tr> <tr> <td>0 0 1 A A A A A</td> <td>A A A A A A A A</td> <td></td> <td></td> </tr> <tr> <td>1 0 i i i i i i</td> <td>i i i i i i i i</td> <td></td> <td></td> </tr> </table>	15	8	7	0	0 0 1 1 1 0 0 0	A A A i i 0 0 1			0 0 1 A A A A A	A A A A A A A A			1 0 i i i i i i	i i i i i i i i		
15	8	7	0																	
0 0 1 1 1 0 0 0	A A A i i 0 0 1																			
0 0 1 A A A A A	A A A A A A A A																			
1 0 i i i i i i	i i i i i i i i																			
AND.W #u16,(SP+s16)	3	3	3	<table border="1"> <tr> <td>15</td> <td>8</td> <td>7</td> <td>0</td> </tr> <tr> <td>0 0 1 1 1 0 0 0</td> <td>A A A i i 0 1 1</td> <td></td> <td></td> </tr> <tr> <td>0 0 1 A A A A A</td> <td>A A A A A A A A</td> <td></td> <td></td> </tr> <tr> <td>1 0 i i i i i i</td> <td>i i i i i i i i</td> <td></td> <td></td> </tr> </table>	15	8	7	0	0 0 1 1 1 0 0 0	A A A i i 0 1 1			0 0 1 A A A A A	A A A A A A A A			1 0 i i i i i i	i i i i i i i i		
15	8	7	0																	
0 0 1 1 1 0 0 0	A A A i i 0 1 1																			
0 0 1 A A A A A	A A A A A A A A																			
1 0 i i i i i i	i i i i i i i i																			

Instruction Fields

Rn	RRR	Address Register					
	000 R0	010 R2	100 R4	110 R6			
	001 R1	011 R3	101 R5	111 R7			

Note: This instruction can specify R8-R15 as operands by using a high register prefix.

a16	AAAAAAAAAAAAAAAA	16-bit unsigned absolute address
#u16	~iiiiiiiiiiiiiii	One's complement of unsigned 16-bit immediate data
u5	AAAAA0	Unsigned 5-bit SP address offset
s16	AAAAAAAAAAAAAAAA	Signed 16-bit SP address offset

Operation

Da << 1 → Dn

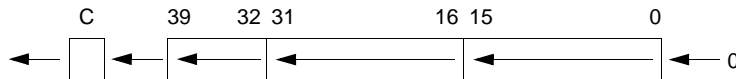
Assembler Syntax

ASL Da, Dn

Description

ASL Da, Dn

Shifts a source data register (Da) left one bit and stores the result in a destination data register (Dn). If the source and destination registers are the same, the original value is destroyed, leaving the shifted value in the register.



Note: The ASL instruction is mapped by the assembler to ADD Da,Db,Dn if Da is an even numbered data register and ADD Da,Da,Dn if Da is an odd numbered data register.

Status and Conditions that Affect Instruction

Register Address	Bit Name	Description
SR[2]	SM	If set, selects 32-bit arithmetic saturation mode.
SR[5:4]	S[1:0]	Scaling mode bits determine which bits in the result are used in the Ln bit calculation.

Status and Conditions Changed by Instruction

Register Address	Bit Name	Description
SR[0]	C	Bit Da[39] is stored in the carry bit.
Ln	L	If not in arithmetic saturation mode (SR [SM] = 0), calculates and updates the Ln bit in the destination register. If in arithmetic saturation mode (SR [SM] = 1), clears the Ln bit in the destination register.
EMR[2]	DOVF	Set if the result cannot be represented in 40 bits, or if the result saturates to 32 bits in arithmetic saturation mode.

Example

```
asl d0,d1
```

Register/Memory Address	Before	After
D0	\$ff f001 0001	
L1:D1		\$0:\$ff e002 0002



Register/Memory Address	Before	After
SR	\$00E0 0000	\$00E0 0001
EMR		\$0000 0000

Instruction Formats and Opcodes

Instruction	Words	Cycles	Type	Opcode																																								
ASL Da, Dn	1	1	1	<table> <tr> <td>15</td> <td>8</td> <td>7</td> <td>0</td> </tr> <tr> <td>0</td> <td>*</td> <td>1</td> <td>0</td> </tr> <tr> <td>1</td> <td>1</td> <td>1</td> <td>F</td> </tr> <tr> <td>F</td> <td>F</td> <td>F</td> <td>1</td> </tr> <tr> <td></td> <td></td> <td></td> <td>0</td> </tr> <tr> <td></td> <td></td> <td></td> <td>J</td> </tr> <tr> <td></td> <td></td> <td></td> <td>J</td> </tr> <tr> <td></td> <td></td> <td></td> <td>J</td> </tr> <tr> <td></td> <td></td> <td></td> <td>J</td> </tr> <tr> <td></td> <td></td> <td></td> <td>J</td> </tr> </table>	15	8	7	0	0	*	1	0	1	1	1	F	F	F	F	1				0				J				J				J				J				J
15	8	7	0																																									
0	*	1	0																																									
1	1	1	F																																									
F	F	F	1																																									
			0																																									
			J																																									
			J																																									
			J																																									
			J																																									
			J																																									
ASL Da, Dn	1	1	1	<table> <tr> <td>15</td> <td>8</td> <td>7</td> <td>0</td> </tr> <tr> <td>0</td> <td>*</td> <td>1</td> <td>0</td> </tr> <tr> <td>0</td> <td>0</td> <td>0</td> <td>F</td> </tr> <tr> <td>F</td> <td>F</td> <td>F</td> <td>1</td> </tr> <tr> <td></td> <td></td> <td></td> <td>1</td> </tr> <tr> <td></td> <td></td> <td></td> <td>0</td> </tr> <tr> <td></td> <td></td> <td></td> <td>0</td> </tr> <tr> <td></td> <td></td> <td></td> <td>0</td> </tr> <tr> <td></td> <td></td> <td></td> <td>j</td> </tr> <tr> <td></td> <td></td> <td></td> <td>j</td> </tr> </table>	15	8	7	0	0	*	1	0	0	0	0	F	F	F	F	1				1				0				0				0				j				j
15	8	7	0																																									
0	*	1	0																																									
0	0	0	F																																									
F	F	F	1																																									
			1																																									
			0																																									
			0																																									
			0																																									
			j																																									
			j																																									

Note: ** indicates serial grouping encoding.

Instruction Fields

Da,Db	JJJJ	Data Register Pairs					
0000	D0,D4	01000	D2,D4	10000	D0,D0	11000	D1,D2
00001	D0,D5	01001	D2,D5	10001	D0,D1	11001	D1,D3
00010	D0,D6	01010	D2,D6	10010	D0,D2	11010	D5,D6
00011	D0,D7	01011	D2,D7	10011	D0,D3	11011	D5,D7
00100	D1,D4	01100	D3,D4	10100	D4,D4	11100	D2,D2
00101	D1,D5	01101	D3,D5	10101	D4,D5	11101	D2,D3
00110	D1,D6	01110	D3,D6	10110	D4,D6	11110	D6,D6
00111	D1,D7	01111	D3,D7	10111	D4,D7	11111	D6,D7

- Notes:**
1. This instruction can specify D8-D15 as operands by using a prefix. Each register of a pair can be separately encoded as the higher register. For example, D0,D4 can be changed to D8,D4 by use of a prefix.
 2. Register pair order can be reversed for clarity because the order of operation is not important for add operations.
 3. The JJJJ encoding does not include the pairs: D1,D1; D3,D3; D5,D5; D7,D7. These are covered in the jj encoding.

Da,Da	jj	Data Register Pairs					
00	D1,D1	01	D3,D3	10	D5,D5	11	D7,D7

Note: This instruction can specify D8-D15 as operands by using a prefix.

Dn	FFF	Single Source/Destination Data Register					
000	D0	010	D2	100	D4	110	D6
001	D1	011	D3	101	D5	111	D7

Note: This instruction can specify D8-D15 as operands by using a prefix.

Operation

$Rx \ll 2 \rightarrow Rx$

Assembler Syntax

ASL2A Rx

Description

ASL2A Rx

Shifts an AGU register (Rx) left two bits. Bits [29:0] are copied into bits [31:2]. Bits [1:0] are cleared.

Status and Conditions that Affect Instruction

Register Address	Bit Name	Description
SR[18]	EXP	Determines which stack pointer is used when the stack pointer is an operand. Otherwise, the instruction is not affected by SR.

Status and Conditions Changed by Instruction

None.

Example

```
asl2a r0
```

Register/Memory Address	Before	After
R0	\$e001 0002	\$8004 0008

Instruction Formats and Opcodes

Instruction	Words	Cycles	Type	Opcode
ASL2A Rx	1	1	2	<div style="display: flex; justify-content: space-between; align-items: center;"> 15 8 7 0 </div> <div style="border: 1px solid black; padding: 2px; display: flex; justify-content: space-between; align-items: center;"> 1 1 1 0 R R R R 1 1 1 1 1 1 1 0 </div>

Instruction Fields

Rx	RRRR	AGU Source/Destination Register					
0000	N0	0100	—	1000	R0	1100	R4
0001	N1	0101	—	1001	R1	1101	R5
0010	N2	0110	—	1010	R2	1110	R6
0011	N3	0111	SP	1011	R3	1111	R7

Note: This instruction can specify R8-R15 as operands by using a high register prefix.



Operation

$Rx \ll 1 \rightarrow Rx$

Assembler Syntax

ASLA Rx

Description

ASLA Rx

Shifts an AGU (Rx) register left one bit. Bits [30:0] are copied into bits [31:1]. Bit 0 is cleared.

Status and Conditions that Affect Instruction

Register Address	Bit Name	Description
SR[18]	EXP	Determines which stack pointer is used when the stack pointer is an operand. Otherwise, the instruction is not affected by SR.

Status and Conditions Changed by Instruction

None.

Example

```
asla r0
```

Register/Memory Address	Before	After
R0	\$e001 0002	\$c002 0004

Instruction Formats and Opcodes

Instruction	Words	Cycles	Type	Opcode
ASLA Rx	1	1	2	<div style="display: flex; justify-content: space-between; align-items: center;"> 15 8 7 0 </div> <div style="border: 1px solid black; padding: 2px; display: flex; justify-content: space-between; align-items: center;"> 1 1 1 0 R R R R 1 1 1 1 1 1 0 0 </div>

Instruction Fields

Rx	RRRR	AGU Source/Destination Register					
0000	N0	0100	—	1000	R0	1100	R4
0001	N1	0101	—	1001	R1	1101	R5
0010	N2	0110	—	1010	R2	1110	R6
0011	N3	0111	SP	1011	R3	1111	R7

Note: This instruction can specify R8-R15 as operands by using a high register prefix.

Operation

$D_n \ll \#u5 \rightarrow D_n$

If $Da[6:0] > 0$, then $D_n \ll Da[6:0] \rightarrow D_n$
 else $D_n \gg |Da[6:0]| \rightarrow D_n$

Assembler Syntax

ASLL #u5, Dn { $0 \leq u5 < 32$ }

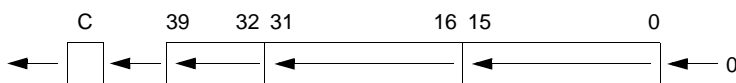
ASLL Da, Dn { $-40 \leq Da[6:0] \leq 40$ }

Description

These operations shift the contents of Dn by the amount in #u5 or in Da. Bits shifted out of Dn are lost except for the last bit, which is stored in the C bit.

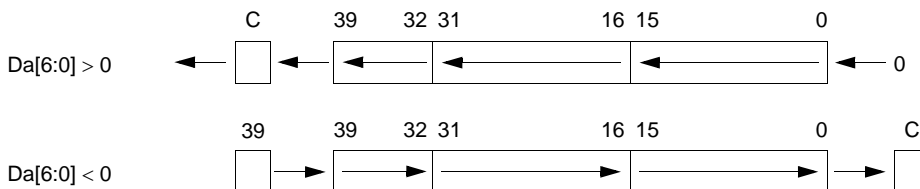
ASLL #u5, Dn

Shifts left by #u5, an immediate unsigned 5-bit integer. The vacated positions to the right are zero-filled.



ASLL Da, Dn

Performs a bidirectional arithmetic shift of Dn by Da[6:0] bits and stores the result in Dn. If Da[6:0] is positive, the shift is left. If shifting left, the vacated positions to the right are zero-filled. If Da[6:0] is negative, the shift is right. If shifting right, the MSB of the source is copied into the vacated positions, creating a sign-extension.



Status and Conditions that Affect Instruction

None.

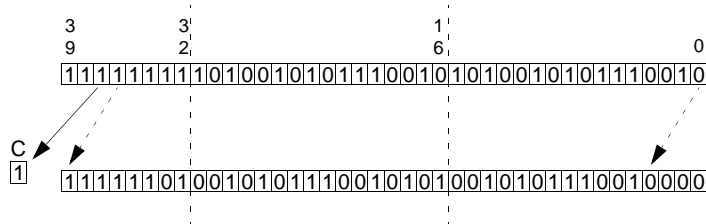
Status and Conditions Changed by Instruction

Register Address	Bit Name	Description
SR[0]	C	Calculates and updates the carry bit in the status register.
EMR[2]	DOVF	Set if the result cannot be represented in 40 bits.
Ln	L	Clears the Ln bit in the destination register.

Example 1

```
asll d0,d1
```

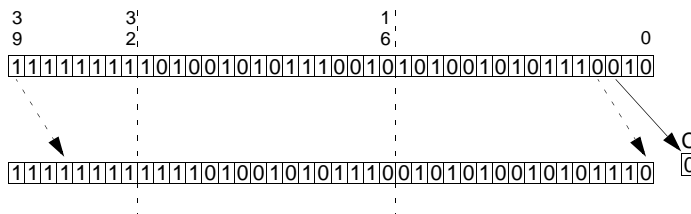
Register/Memory Address	Before	After
D0	\$00 0000 0003	
L1:D1	\$0:\$FF A572 A572	\$0:\$FD 2B95 2B90
SR	\$00E0 0000	\$00E0 0001
EMR		\$0000 0000



Example 2

```
asll d0,d1
```

Register/Memory Address	Before	After
D0	\$FF FFFF FF0D	
L1:D1	\$0:\$FF A572 A572	\$0:\$FF F4AE 54AE
SR	\$00E4 0000	\$00E4 0000
EMR		\$0000 0000





ASLW

Word Arithmetic Shift Left 16 Bits (DALU)

ASLW

Operation

$Da \ll 16 \rightarrow Dn$

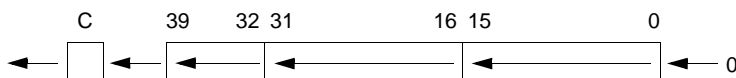
Assembler Syntax

ASLW Da, Dn

Description

ASLW Da, Dn

Shifts the source register Da left by 16 bits and stores it in the destination register Dn. Bit 24 of the source register is copied into the C bit. Bits [23:0] of the source register are copied into bits [39:16] of the destination register. Bits [15:0] of the destination register are cleared.



Status and Conditions that Affect Instruction

None.

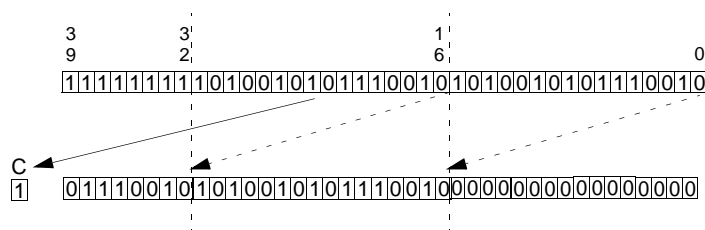
Status and Conditions Changed by Instruction

Register Address	Bit Name	Description
SR[0]	C	Bit Da[24] is stored in the carry bit.
EMR[2]	DOVF	Set if the result cannot be represented in 40 bits.
Ln	L	Clears the Ln bit in the destination register.

Example

aslw d0, d1

Register/Memory Address	Before	After
D0	\$FF A572 A572	
L1:D1	\$0:\$00 0000 0000	\$0:\$72 A572 0000
SR	\$00E0 0000	\$00E0 0001
EMR		\$0000 0004





Operation

$Da \gg 1 \rightarrow Dn$

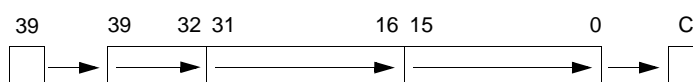
Assembler Syntax

ASR Da, Dn

Description

ASR Da, Dn

Performs an arithmetic right shift by one bit on the source register Da , and stores it in the destination register Dn . The LSB (bit 0) of the source register is copied into the status register carry (C) bit. Bits [39:1] of the source register are copied into bits [38:0] of the destination register. Bit 39 of the source register is copied into bit 39 of the destination register.



Status and Conditions that Affect Instruction

Register Address	Bit Name	Description
SR[2]	SM	If set, selects 32-bit arithmetic saturation mode.
SR[5:4]	S[1:0]	Scaling mode bits determine which bits in the result are used in the L_n bit calculation.

Status and Conditions Changed by Instruction

Register Address	Bit Name	Description
SR[0]	C	$Da[0]$ is stored in the carry bit.
L_n	L	If not in arithmetic saturation mode (SR [SM] = 0), calculates and updates the L_n bit in the destination register. If in arithmetic saturation mode (SR [SM] = 1), clears the L_n bit in the destination register.
EMR[2]	DOVF	Set if the result cannot be represented in 40 bits, or if the result saturates to 32 bits in arithmetic saturation mode.

Example

asr d5, d3

Register/Memory Address	Before	After
D5	\$00 0000 7903	
L3:D3		\$0:\$00 0000 3C81

Register/Memory Address	Before	After
SR	\$00E4 0000	\$00E4 0001
EMR		\$0000 0000

Instruction Formats and Opcodes

Instruction	Words	Cycles	Type	Opcode
ASR Da, Dn	1	1	1	<div style="display: flex; justify-content: space-between; align-items: center;"> 15 8 7 0 </div> <div style="border: 1px solid black; padding: 2px; display: flex; justify-content: space-between; align-items: center;"> 0 * 1 1 0 1 F F F 1 0 0 0 J J J </div>

Note: ** indicates serial grouping encoding.

Instruction Fields

Da **JJJ** **Single Source Data Register**

000	D0	010	D2	100	D4	110	D6
001	D1	011	D3	101	D5	111	D7

Note: This instruction can specify D8-D15 as operands by using a prefix.

Dn **FFF** **Single Source/Destination Data Register**

000	D0	010	D2	100	D4	110	D6
001	D1	011	D3	101	D5	111	D7

Note: This instruction can specify D8-D15 as operands by using a prefix.



Operation

$Rx \gg 1 \rightarrow Rx$

Assembler Syntax

ASRA Rx

Description

ASRA Rx

Performs an arithmetic right shift by one bit on the AGU register (Rx). Moves bits [31:1] into bits [30:0]. Bit 31 remains the same, creating a sign-extension.

Status and Conditions that Affect Instruction

Register Address	Bit Name	Description
SR[18]	EXP	Determines which stack pointer is used when the stack pointer is an operand. Otherwise, the instruction is not affected by SR.

Status and Conditions Changed by Instruction

None.

Example

asra r2

Register/Memory Address	Before	After
R2	\$8002 0002	\$C001 0001

Instruction Formats and Opcodes

Instruction	Words	Cycles	Type	Opcode
ASRA Rx	1	1	2	<div style="display: flex; justify-content: space-between; align-items: center;"> 15 8 7 0 </div> <div style="border: 1px solid black; padding: 2px; display: flex; justify-content: space-between; align-items: center;"> 1 1 1 0 R R R R 1 1 1 1 1 1 0 1 </div>

Instruction Fields

Rx RRRR AGU Source/Destination Register

0000	N0	0100	—	1000	R0	1100	R4
0001	N1	0101	—	1001	R1	1101	R5
0010	N2	0110	—	1010	R2	1110	R6
0011	N3	0111	SP	1011	R3	1111	R7

Note: This instruction can specify R8-R15 as operands by using a high register prefix.

Operation

$D_n \gg \#u5 \rightarrow D_n$

If $Da[6:0] > 0$, then $D_n \gg Da[6:0] \rightarrow D_n$
 else $D_n \ll |Da[6:0]| \rightarrow D_n$

Assembler Syntax

ASRR #u5, Dn { $0 \leq u5 < 32$ }

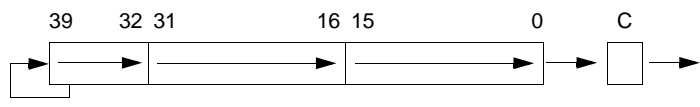
ASRR Da, Dn

Description

This operation shifts the contents of Dn by N bits. Bits shifted out of Dn are lost except for the last bit, which is stored in the C bit.

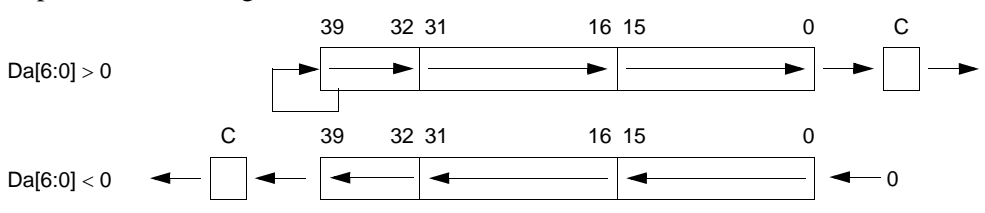
ASRR #u5, Dn

Performs an arithmetic right shift by N, an immediate unsigned 5-bit integer. The MSB is copied into the vacated positions.



ASRR Da, Dn

Performs a bidirectional arithmetic shift of Dn by Da[6:0] bits and stores the result in Dn. If Da[6:0] is positive, the shift is right. If shifting right, the MSB is copied into the vacated positions. If shifting left, the vacated positions to the right are zero-filled. N is obtained from Da[6:0].



Status and Conditions that Affect Instruction

None.

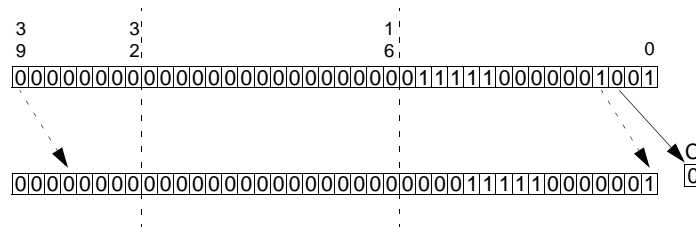
Status and Conditions Changed by Instruction

Register Address	Bit Name	Description
SR[0]	C	Set to the value of the last bit out.
EMR[2]	DOVF	Set if the result cannot be represented in 40 bits (possible only for ASRR Da,Dn)
Ln	L	Clears the Ln bit in the destination register.

Example 1

```
asrr # $3, d5
```

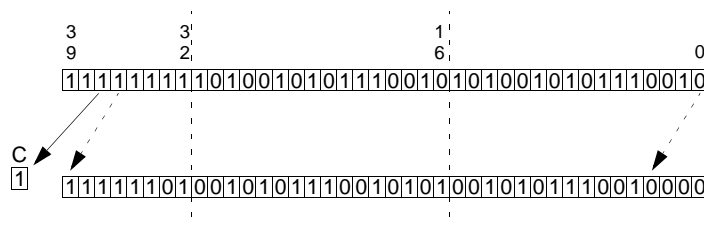
Register/Memory Address	Before	After
D3	\$3	
L5:D5	\$0:\$00 0000 7C09	\$0:\$00 0000 0F81
SR	\$00E4 0000	\$00E4 0000
EMR		\$0000 0000



Example 2

```
asrr d3, d5
```

Register/Memory Address	Before	After
D3	\$FF FDDD DDDC	
L5:D5	\$0:\$00 0000 7C09	\$0:\$00 0007 C090
SR	\$00E4 0000	\$00E4 0001
EMR		\$0000 0000



Instruction Formats and Opcodes

Instruction	Words	Cycles	Type	Opcode																
ASRR #u5, Dn	1	1	1	<div style="display: flex; justify-content: space-between; align-items: center;"> 15 8 7 0 </div> <table border="1" style="width: 100%; border-collapse: collapse;"> <tr> <td style="width: 10%;">0</td> <td style="width: 10%;">*</td> <td style="width: 10%;">1</td> <td style="width: 10%;">1</td> <td style="width: 10%;">1</td> <td style="width: 10%;">1</td> <td style="width: 10%;">1</td> <td style="width: 10%;">F</td> <td style="width: 10%;">F</td> <td style="width: 10%;">F</td> <td style="width: 10%;">1</td> <td style="width: 10%;">1</td> <td style="width: 10%;">i</td> <td style="width: 10%;">i</td> <td style="width: 10%;">i</td> <td style="width: 10%;">i</td> </tr> </table>	0	*	1	1	1	1	1	F	F	F	1	1	i	i	i	i
0	*	1	1	1	1	1	F	F	F	1	1	i	i	i	i					
ASRR Da, Dn	1	1	2	<div style="display: flex; justify-content: space-between; align-items: center;"> 15 8 7 0 </div> <table border="1" style="width: 100%; border-collapse: collapse;"> <tr> <td style="width: 10%;">1</td> <td style="width: 10%;">1</td> <td style="width: 10%;">0</td> <td style="width: 10%;">1</td> <td style="width: 10%;">0</td> <td style="width: 10%;">1</td> <td style="width: 10%;">F</td> <td style="width: 10%;">F</td> <td style="width: 10%;">F</td> <td style="width: 10%;">0</td> <td style="width: 10%;">0</td> <td style="width: 10%;">1</td> <td style="width: 10%;">1</td> <td style="width: 10%;">J</td> <td style="width: 10%;">J</td> <td style="width: 10%;">J</td> </tr> </table>	1	1	0	1	0	1	F	F	F	0	0	1	1	J	J	J
1	1	0	1	0	1	F	F	F	0	0	1	1	J	J	J					

Note: ** indicates serial grouping encoding.

Instruction Fields

Da **JJJ** **Single Source Data Register**

000	D0	010	D2	100	D4	110	D6
001	D1	011	D3	101	D5	111	D7

Note: This instruction can specify D8-D15 as operands by using a prefix.

Dn **FFF** **Single Source/Destination Data Register**

000	D0	010	D2	100	D4	110	D6
001	D1	011	D3	101	D5	111	D7

Note: This instruction can specify D8-D15 as operands by using a prefix.

#u5 **iiii** 5-bit unsigned immediate data



ASRW

Word Arithmetic Shift Right 16 Bits (DALU)

ASRW

Operation

$Da \gg 16 \rightarrow Dn$

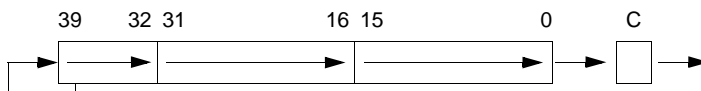
Assembler Syntax

ASRW Da, Dn

Description

ASRW Da, Dn

Performs an arithmetic right shift of 16 bits on the source register Da and stores the result in the destination register Dn. It copies bit 39 of the source register to bits [39:24] of the destination register, bit 15 of the source register to the C bit, and bits [39:16] of the source register to bits [23:0] of the destination register.



Status and Conditions that Affect Instruction

None.

Status and Conditions Changed by Instruction

Register Address	Bit Name	Description
SR[0]	C	Da[15] is stored in the carry bit.
Ln	L	Clears the Ln bit in the destination register.

Example

asrw d5, d0

Register/Memory Address	Before	After
D5	\$80 1234 8765	
L0:D0	\$0:\$00 0000 0000	\$0:\$ff ff80 1234
SR	\$00E0 0000	\$00E0 0001

Instruction Formats and Opcodes

Instruction	Words	Cycles	Type	Opcode																
ASRW Da, Dn	1	1	2	<div style="display: flex; justify-content: space-between; align-items: center;"> 15 8 7 0 </div> <table border="1" style="width: 100%; text-align: center; border-collapse: collapse;"> <tr> <td style="width: 25%;">1</td> <td style="width: 25%;">1</td> <td style="width: 25%;">0</td> <td style="width: 25%;">1</td> <td style="width: 25%;">1</td> <td style="width: 25%;">0</td> <td style="width: 25%;">F</td> <td style="width: 25%;">F</td> <td style="width: 25%;">F</td> <td style="width: 25%;">0</td> <td style="width: 25%;">0</td> <td style="width: 25%;">1</td> <td style="width: 25%;">1</td> <td style="width: 25%;">J</td> <td style="width: 25%;">J</td> <td style="width: 25%;">J</td> </tr> </table>	1	1	0	1	1	0	F	F	F	0	0	1	1	J	J	J
1	1	0	1	1	0	F	F	F	0	0	1	1	J	J	J					

Instruction Fields

Da	JJJ	Single Source Data Register					
000	D0	010	D2	100	D4	110	D6
001	D1	011	D3	101	D5	111	D7

Note: This instruction can specify D8-D15 as operands by using a prefix.

Dn	FFF	Single Source/Destination Data Register					
000	D0	010	D2	100	D4	110	D6
001	D1	011	D3	101	D5	111	D7

Note: This instruction can specify D8-D15 as operands by using a prefix.

Operation

If $T=0$, then $PC + \text{displacement} \rightarrow PC$

Assembler Syntax

```
BF <label
BF >label
```

Description

BF <label

BF >label

Branches to label if the true bit is cleared. If the T bit is cleared, the program continues executing at location $PC + \text{displacement}$. If the T bit is set, the PC is updated to point to the next execution set, and the program continues executing sequentially. The displacement, calculated by the assembler and linker, is a two's complement integer that represents the relative distance from the current PC to the destination label. The assembler determines if the PC relative displacement is a short branch ($\text{<label} [-2^8 \leq \text{displacement} < 2^8, W]$) or a long branch ($\text{>label} [-2^{20} \leq \text{displacement} < -2^8, W \text{ and } 2^8 \leq \text{displacement} < 2^{20}, W]$).

Status and Conditions that Affect Instruction

Register Address	Bit Name	Description
SR[1]	T	True bit

Status and Conditions Changed by Instruction

None.

Example

```
BF lbl
```

Instruction	Result
<code>cmpeq.w #35,d1</code>	Not equal, so T bit in SR cleared.
<code>bf lbl move.w #29,d1</code>	Branch taken, <code>move.w</code> executed.
<code>inc d1</code>	Skipped over.
<code>move.w #47,d2</code>	Skipped over.
<code>- - - -</code>	Skipped over.
<code>- - - -</code>	Skipped over.
<code>- - - -</code>	Skipped over.
<code>lbl move.w #16,d4</code>	Execution continues here at <code>lbl</code> .

Register/Memory Address	Before	After
SR	\$00E4 0000	
d1	\$0000	\$0029

Register/Memory Address	Before	After
d2	\$0000	\$0000
pc	\$0006	\$0016

Instruction Formats and Opcodes

Instruction	Words	Cycles ¹	Type	Opcode																																				
BF <label	1	1/4	4	<table border="1"> <tr> <td>15</td> <td>8</td> <td>7</td> <td>0</td> </tr> <tr> <td>1</td><td>0</td><td>0</td><td>0</td> <td>0</td><td>0</td><td>1</td><td>0</td> <td>A</td><td>A</td><td>A</td><td>A</td><td>A</td><td>A</td><td>A</td><td>1</td> </tr> </table>	15	8	7	0	1	0	0	0	0	0	1	0	A	A	A	A	A	A	A	1																
15	8	7	0																																					
1	0	0	0	0	0	1	0	A	A	A	A	A	A	A	1																									
BF >label	2	1/4	4	<table border="1"> <tr> <td>15</td> <td>8</td> <td>7</td> <td>0</td> </tr> <tr> <td>0</td><td>0</td><td>1</td><td>0</td> <td>a</td><td>1</td><td>1</td><td>1</td> <td>A</td><td>A</td><td>A</td><td>1</td><td>1</td> <td>a</td><td>a</td><td>a</td> </tr> <tr> <td>1</td><td>0</td><td>0</td><td>A</td> <td>A</td><td>A</td><td>A</td><td>A</td> <td>A</td><td>A</td><td>A</td><td>A</td><td>A</td><td>A</td><td>A</td><td>a</td> </tr> </table>	15	8	7	0	0	0	1	0	a	1	1	1	A	A	A	1	1	a	a	a	1	0	0	A	A	A	A	A	A	A	A	A	A	A	A	a
15	8	7	0																																					
0	0	1	0	a	1	1	1	A	A	A	1	1	a	a	a																									
1	0	0	A	A	A	A	A	A	A	A	A	A	A	A	a																									

Note 1: If the branch is not taken, it uses 1 cycle. If the branch is taken, it uses 4 cycles.

Instruction Fields

displacement (<label)	AAAAAAAAA0	8-bit signed PC relative displacement
displacement (>label)	aaaaaAAAAAAAAAAAAAAAAA0	20-bit signed PC relative displacement

Operation

If $T=0$, then $PC + displacement \rightarrow PC$

Assembler Syntax

```
BFD <label
BFD >label
```

Description

BFD <label

BFD >label

Branches to label if the true bit is cleared. If the T bit is cleared, the program continues executing at location $PC + displacement$. If the T bit is set, the PC is updated to point to the next execution set, and the program continues executing sequentially. The displacement, calculated by the assembler and linker, is a two's complement integer that represents the relative distance from the current PC to the destination label. The assembler determines if the PC relative displacement is a short branch (<label [$-2^8 \leq displacement < 2^8$, W]) or a long branch (>label [$-2^{20} \leq displacement < -2^8$, W and $2^8 \leq displacement < 2^{20}$, W]). The execution set in the delay slot immediately following the BFD instruction is executed unconditionally after the execution set containing the BFD instruction.

Status and Conditions that Affect Instruction

Register Address	Bit Name	Description
SR[1]	T	True bit

Status and Conditions Changed by Instruction

None.

Example

BFD lbl

Instruction	Result
cmpeq.w #35,d1	Not equal, so T bit in SR cleared.
bfd lbl move.w #29,d1	Branch taken, move.w executed.
inc d1	Increment executed in the delay slot.
move.w #47,d2	Skipped over.
- - - -	Skipped over.
- - - -	Skipped over.
- - - -	Skipped over.
lbl move.w #1A,d4	Execution continues here at lbl.

Register/Memory Address	Before	After
SR	\$00E0 0000	

Register/Memory Address	Before	After
d1	\$0000	\$002A
d2	\$0000	\$0000
d4	\$0000	\$001A
pc	\$0006	\$0016

Instruction Formats and Opcodes

Instruction	Words	Cycles ¹	Type	Opcode												
BFD <label	1	1/4	4	<table border="1"> <tr> <td style="text-align: center;">15</td> <td style="text-align: center;">8</td> <td style="text-align: center;">7</td> <td style="text-align: center;">0</td> </tr> <tr> <td>1 0 0 0 0 1 0 A</td> <td>A</td> <td>A</td> <td>A A A A A A A 0</td> </tr> </table>	15	8	7	0	1 0 0 0 0 1 0 A	A	A	A A A A A A A 0				
15	8	7	0													
1 0 0 0 0 1 0 A	A	A	A A A A A A A 0													
BFD >label	2	1/4	4	<table border="1"> <tr> <td style="text-align: center;">15</td> <td style="text-align: center;">8</td> <td style="text-align: center;">7</td> <td style="text-align: center;">0</td> </tr> <tr> <td>0 0 1 0 a 1 1 0</td> <td>A</td> <td>A A 1 1 a a a</td> <td></td> </tr> <tr> <td>1 0 0 A A A A A</td> <td>A</td> <td>A A A A A A a</td> <td></td> </tr> </table>	15	8	7	0	0 0 1 0 a 1 1 0	A	A A 1 1 a a a		1 0 0 A A A A A	A	A A A A A A a	
15	8	7	0													
0 0 1 0 a 1 1 0	A	A A 1 1 a a a														
1 0 0 A A A A A	A	A A A A A A a														

Note 1: If the branch is not taken, it uses 1 cycle. If the branch is taken, it uses 4 cycles minus the time used by the execution set in the delay slot. The cycle count for this instruction cannot be less than 1 cycle.

Instruction Fields

displacement (<label)	AAAAAAAA0	8-bit signed PC relative displacement
displacement (>label)	aaaaaAAAAAAAAAAAAAAAAA0	20-bit signed PC relative displacement



Operation

$\sim C1.H_i \rightarrow C1.H_i$ (i denotes bits=1 in #u16)

$\sim C1.L_i \rightarrow C1.L_i$

$\sim DR.H_i \rightarrow DR.H_i$

$\sim DR.L_i \rightarrow DR.L_i$

Assembler Syntax

BMCHG #u16,C1.H $\{0 \leq u16 < 2^{16}\}$

BMCHG #u16,C1.L $\{0 \leq u16 < 2^{16}\}$

BMCHG #u16,DR.H $\{0 \leq u16 < 2^{16}\}$

BMCHG #u16,DR.L $\{0 \leq u16 < 2^{16}\}$

Description

These operations use an unsigned 16-bit immediate data mask to invert selected bits in the destination operand. For each bit i that is set (selected) in the mask, the bit i in the corresponding destination operand's bit position is inverted. Bits that are not selected as well as bits in the other part of the register are unaffected. These operations read from a register, modify the retrieved value, and write the new value back to that register. The operation is equivalent to the exclusive-or function.

Note: Special care must be taken when using this instruction to clear bits on the EMR register due to this register's special functionality. See Chapter 3 for a description of this behavior.

BMCHG

#u16,C1.H

Inverts selected bits in the contents of the HP of a control register (C1).

BMCHG

#u16,C1.L

Inverts selected bits in the contents of the LP of a control register (C1).

BMCHG

#u16,DR.H

Inverts selected bits in the contents of the HP of a data or address register (DR).

BMCHG

#u16,DR.L

Inverts selected bits in the contents of the LP of a data or address register (DR).

Status and Conditions that Affect Instruction

Register Address	Bit Name	Description
SR[18]	EXP	Determines execution working mode for instructions that have these registers as an operand.

Status and Conditions Changed by Instruction

Register Address	Bit Name	Description
Ln	L	Clears the Ln bit in the destination data register.

Example

```
bmchg #f0f0,d1.h
```

Register/Memory Address	Before	After
immediate	\$F0F00000	
L1:D1	\$0:\$FFF0F07B22	\$0:\$FF00007B22

Instruction Formats and Opcodes

Instruction	Words	Cycles	Type	Opcode																																						
BMCHG #u16,C1.H	2	2	3	<table border="1"> <tr> <td>15</td> <td></td> <td>8</td> <td>7</td> <td></td> <td>0</td> </tr> <tr> <td>0</td><td>0</td><td>0</td><td>1</td><td>0</td><td>0</td><td>1</td><td>0</td> <td>i</td><td>i</td><td>i</td><td>1</td><td>0</td><td>C</td><td>C</td><td>C</td> </tr> <tr> <td>1</td><td>0</td><td>1</td><td>i</td><td>i</td><td>i</td><td>i</td><td>i</td> <td>i</td><td>i</td><td>i</td><td>i</td><td>i</td><td>i</td><td>i</td><td>i</td> </tr> </table>	15		8	7		0	0	0	0	1	0	0	1	0	i	i	i	1	0	C	C	C	1	0	1	i	i	i	i	i	i	i	i	i	i	i	i	i
15		8	7		0																																					
0	0	0	1	0	0	1	0	i	i	i	1	0	C	C	C																											
1	0	1	i	i	i	i	i	i	i	i	i	i	i	i	i																											
BMCHG #u16,C1.L	2	2	3	<table border="1"> <tr> <td>15</td> <td></td> <td>8</td> <td>7</td> <td></td> <td>0</td> </tr> <tr> <td>0</td><td>0</td><td>0</td><td>1</td><td>0</td><td>0</td><td>1</td><td>0</td> <td>i</td><td>i</td><td>i</td><td>0</td><td>0</td><td>C</td><td>C</td><td>C</td> </tr> <tr> <td>1</td><td>0</td><td>1</td><td>i</td><td>i</td><td>i</td><td>i</td><td>i</td> <td>i</td><td>i</td><td>i</td><td>i</td><td>i</td><td>i</td><td>i</td><td>i</td> </tr> </table>	15		8	7		0	0	0	0	1	0	0	1	0	i	i	i	0	0	C	C	C	1	0	1	i	i	i	i	i	i	i	i	i	i	i	i	i
15		8	7		0																																					
0	0	0	1	0	0	1	0	i	i	i	0	0	C	C	C																											
1	0	1	i	i	i	i	i	i	i	i	i	i	i	i	i																											
BMCHG #u16,DR.H	2	2	3	<table border="1"> <tr> <td>15</td> <td></td> <td>8</td> <td>7</td> <td></td> <td>0</td> </tr> <tr> <td>0</td><td>0</td><td>0</td><td>0</td><td>1</td><td>0</td><td>1</td><td>0</td> <td>i</td><td>i</td><td>i</td><td>1</td><td>H</td><td>H</td><td>H</td><td>H</td> </tr> <tr> <td>1</td><td>0</td><td>1</td><td>i</td><td>i</td><td>i</td><td>i</td><td>i</td> <td>i</td><td>i</td><td>i</td><td>i</td><td>i</td><td>i</td><td>i</td><td>i</td> </tr> </table>	15		8	7		0	0	0	0	0	1	0	1	0	i	i	i	1	H	H	H	H	1	0	1	i	i	i	i	i	i	i	i	i	i	i	i	i
15		8	7		0																																					
0	0	0	0	1	0	1	0	i	i	i	1	H	H	H	H																											
1	0	1	i	i	i	i	i	i	i	i	i	i	i	i	i																											
BMCHG #u16,DR.L	2	2	3	<table border="1"> <tr> <td>15</td> <td></td> <td>8</td> <td>7</td> <td></td> <td>0</td> </tr> <tr> <td>0</td><td>0</td><td>0</td><td>0</td><td>1</td><td>0</td><td>1</td><td>0</td> <td>i</td><td>i</td><td>i</td><td>0</td><td>H</td><td>H</td><td>H</td><td>H</td> </tr> <tr> <td>1</td><td>0</td><td>1</td><td>i</td><td>i</td><td>i</td><td>i</td><td>i</td> <td>i</td><td>i</td><td>i</td><td>i</td><td>i</td><td>i</td><td>i</td><td>i</td> </tr> </table>	15		8	7		0	0	0	0	0	1	0	1	0	i	i	i	0	H	H	H	H	1	0	1	i	i	i	i	i	i	i	i	i	i	i	i	i
15		8	7		0																																					
0	0	0	0	1	0	1	0	i	i	i	0	H	H	H	H																											
1	0	1	i	i	i	i	i	i	i	i	i	i	i	i	i																											

Instruction Fields

C1	CCC	Control Registers					
000	EMR	010	-	100	—	110	—
001	VBA	011	-	101	SR	111	MCTL

DR	HHHH	Data/Address Register					
0000	D0	0100	D4	1000	R0	1100	R4
0001	D1	0101	D5	1001	R1	1101	R5
0010	D2	0110	D6	1010	R2	1110	R6
0011	D3	0111	D7	1011	R3	1111	R7

Note: If registers D8–D15 or R8–R15 are accessed instead of D0–D7 or R0–R7, a prefix is used.

#u16

iiiiiiiiiiiiii 16-bit unsigned immediate data



BMCHG.W

Bit-Masked Change a 16-Bit Operand in Memory (BMU)

BMCHG.W

Operation

$\sim(\text{SP}-u5)_i \rightarrow (\text{SP}-u5)_i$
(i denotes bits=1 in #u16)

$\sim(\text{SP}+s16)_i \rightarrow (\text{SP}+s16)_i$

$\sim(\text{Rn})_i \rightarrow (\text{Rn})_i$

$\sim(a16)_i \rightarrow (a16)_i$

Assembler Syntax

`BMCHG.W #u16, (SP-u5) {0 ≤ u16 < 216} {0 ≤ u5 < 64, W}`

`BMCHG.W #u16, (SP+s16) {0 ≤ u16 < 216} {-215 ≤ s16 < 215, W}`

`BMCHG.W #u16, (Rn) {0 ≤ u16 < 216}`

`BMCHG.W #u16, (a16) {0 ≤ u16 < 216} {0 ≤ a16 < 216, W}`

Description

These operations use an unsigned 16-bit immediate data mask to invert selected bits in the destination operand. For each bit *i* that is set (selected) in the mask, the bit *i* in the corresponding destination operation's bit position is inverted. These operations read from memory, modify the retrieved value, and write the new value back to that memory address, resulting in two memory accesses. The absolute addresses, offsets, and address register values must be word-aligned.

BMCHG.W #u16,(SP-u5)

Inverts selected bits in the contents of a memory address pointed to by the active stack pointer (SP) with an unsigned 5-bit offset.

BMCHG.W #u16,(SP+s16)

Inverts selected bits in the contents of a memory address pointed to by the active stack pointer (SP) with a signed 16-bit offset.

BMCHG.W #u16,(Rn)

Inverts selected bits in the contents of a memory address pointed to by an address register (Rn).

BMCHG.W #u16,(a16)

Inverts selected bits in the contents of a memory address pointed to by an absolute 16-bit address.

Status and Conditions that Affect Instruction

Register Address	Bit Name	Description
SR[18]	EXP	Determines which stack pointer is used when the stack pointer is an operand. Otherwise, the instruction is not affected by SR.

Status and Conditions Changed by Instruction

None.

xample

bmchg.w # $\$661f$, < $\$800c$

Register/Memory Address	Before	After
immediate	$\$661F$	
$\$800C$	$\$ACE1$	$\$CAFE$
In binary, $\$661F$	0110011000011111	
$\$ACE1$	1010110011100001	
$\$CAFE$	1100101011111110	

Instruction Formats and Opcodes

Instruction	Words	Cycles	Type	Opcode																
BMCHG.W #u16, (SP-u5)	2	3	3	<table border="1"> <tr> <td>15</td> <td>8</td> <td>7</td> <td>0</td> </tr> <tr> <td>0 0 0 0 0 0 1 0</td> <td>i i i A A A A A</td> <td></td> <td></td> </tr> <tr> <td>1 0 1 i i i i i i</td> <td>i i i i i i i i</td> <td></td> <td></td> </tr> </table>	15	8	7	0	0 0 0 0 0 0 1 0	i i i A A A A A			1 0 1 i i i i i i	i i i i i i i i						
15	8	7	0																	
0 0 0 0 0 0 1 0	i i i A A A A A																			
1 0 1 i i i i i i	i i i i i i i i																			
BMCHG.W #u16, (SP+s16)	3	3	3	<table border="1"> <tr> <td>15</td> <td>8</td> <td>7</td> <td>0</td> </tr> <tr> <td>0 0 1 1 1 0 1 0</td> <td>A A A i i 0 1 1</td> <td></td> <td></td> </tr> <tr> <td>0 0 1 A A A A A</td> <td>A A A A A A A A</td> <td></td> <td></td> </tr> <tr> <td>1 0 i i i i i i i</td> <td>i i i i i i i i</td> <td></td> <td></td> </tr> </table>	15	8	7	0	0 0 1 1 1 0 1 0	A A A i i 0 1 1			0 0 1 A A A A A	A A A A A A A A			1 0 i i i i i i i	i i i i i i i i		
15	8	7	0																	
0 0 1 1 1 0 1 0	A A A i i 0 1 1																			
0 0 1 A A A A A	A A A A A A A A																			
1 0 i i i i i i i	i i i i i i i i																			
BMCHG.W #u16, (Rn)	2	2	3	<table border="1"> <tr> <td>15</td> <td>8</td> <td>7</td> <td>0</td> </tr> <tr> <td>0 0 0 1 0 0 1 0</td> <td>i i i 0 1 R R R</td> <td></td> <td></td> </tr> <tr> <td>1 0 1 i i i i i i</td> <td>i i i i i i i i</td> <td></td> <td></td> </tr> </table>	15	8	7	0	0 0 0 1 0 0 1 0	i i i 0 1 R R R			1 0 1 i i i i i i	i i i i i i i i						
15	8	7	0																	
0 0 0 1 0 0 1 0	i i i 0 1 R R R																			
1 0 1 i i i i i i	i i i i i i i i																			
BMCHG.W #u16, (a16)	3	2	3	<table border="1"> <tr> <td>15</td> <td>8</td> <td>7</td> <td>0</td> </tr> <tr> <td>0 0 1 1 1 0 1 0</td> <td>A A A i i 0 0 1</td> <td></td> <td></td> </tr> <tr> <td>0 0 1 A A A A A</td> <td>A A A A A A A A</td> <td></td> <td></td> </tr> <tr> <td>1 0 i i i i i i i</td> <td>i i i i i i i i</td> <td></td> <td></td> </tr> </table>	15	8	7	0	0 0 1 1 1 0 1 0	A A A i i 0 0 1			0 0 1 A A A A A	A A A A A A A A			1 0 i i i i i i i	i i i i i i i i		
15	8	7	0																	
0 0 1 1 1 0 1 0	A A A i i 0 0 1																			
0 0 1 A A A A A	A A A A A A A A																			
1 0 i i i i i i i	i i i i i i i i																			

Instruction Fields

Rn	RRR	Address Register					
000	R0	010	R2	100	R4	110	R6
001	R1	011	R3	101	R5	111	R7

Note: This instruction can specify R8-R15 as operands by using a high register prefix.

a16	AAAAAAAAAAAAAAAA	16-bit unsigned absolute address
#u16	iiiiiiiiiiiiiiii	16-bit unsigned immediate data
u5	AAAAA0	5-bit unsigned SP address offset



3.6

AAAAAAAAAAAAAAAA

16-bit signed SP address offset

BMCLR Bit-Masked Clear a 16-Bit Operand (BMU) BMCLR

Operation

$0 \rightarrow C1.H_i$ (i denotes bits=1 in #u16)

$0 \rightarrow C1.L_i$

$0 \rightarrow DR.H_i$

$0 \rightarrow DR.L_i$

Assembler Syntax

BMCLR #u16,C1.H { $0 \leq u16 < 2^{16}$ }

BMCLR #u16,C1.L { $0 \leq u16 < 2^{16}$ }

BMCLR #u16,DR.H { $0 \leq u16 < 2^{16}$ }

BMCLR #u16,DR.L { $0 \leq u16 < 2^{16}$ }

Description

These operations use an unsigned 16-bit immediate data mask to clear selected bits in the destination operand. For each bit i that is set (selected) in the mask, the bit i in the corresponding destination operand's bit position is cleared. Bits that are not selected as well as bits in the other part of the register are unaffected. These operations read from a register, modify the retrieved value, and write the new value back to that register.

Note: Special care must be taken when using this instruction to clear bits on the EMR register due to this register's special functionality. See Chapter 3 for a description of this behavior.

BMCLR #u16,C1.H

Clears selected bits in the HP contents of a control register (C1).

BMCLR #u16,C1.L

Clears selected bits in the LP contents of a control register (C1).

BMCLR #u16,DR.H

Clears selected bits in the HP contents of a data or address register (DR).

BMCLR #u16,DR.L

Clears selected bits in the LP contents of a data or address register (DR).

Status and Conditions that Affect Instruction

Register Address	Bit Name	Description
SR[18]	EXP	Determines execution working mode for instructions that have these registers as an operand.



Status and Conditions Changed by Instruction

Register Address	Bit Name	Description
Ln	L	Clears the Ln bit in the destination register.

Example

```
bmclr #b646,d7.l
```

Register/Memory Address	Before	After
immediate	\$B646	
L7:D7	\$0:\$0050006C5A	\$0:\$0050004818

Instruction Formats and Opcodes

Instruction	Words	Cycles	Type	Opcode												
BMCLR #u16,C1.H	2	2	3	<table border="1"> <tr> <td>15</td> <td>8</td> <td>7</td> <td>0</td> </tr> <tr> <td>0 0 0 1 0 0 0 0</td> <td>i i i 1 0 C C C</td> <td></td> <td></td> </tr> <tr> <td>1 0 1 i i i i i</td> <td>i i i i i i i i</td> <td></td> <td></td> </tr> </table>	15	8	7	0	0 0 0 1 0 0 0 0	i i i 1 0 C C C			1 0 1 i i i i i	i i i i i i i i		
15	8	7	0													
0 0 0 1 0 0 0 0	i i i 1 0 C C C															
1 0 1 i i i i i	i i i i i i i i															
BMCLR #u16,C1.L	2	2	3	<table border="1"> <tr> <td>15</td> <td>8</td> <td>7</td> <td>0</td> </tr> <tr> <td>0 0 0 1 0 0 0 0</td> <td>i i i 0 0 C C C</td> <td></td> <td></td> </tr> <tr> <td>1 0 1 i i i i i</td> <td>i i i i i i i i</td> <td></td> <td></td> </tr> </table>	15	8	7	0	0 0 0 1 0 0 0 0	i i i 0 0 C C C			1 0 1 i i i i i	i i i i i i i i		
15	8	7	0													
0 0 0 1 0 0 0 0	i i i 0 0 C C C															
1 0 1 i i i i i	i i i i i i i i															
BMCLR #u16,DR.H	2	2	3	<table border="1"> <tr> <td>15</td> <td>8</td> <td>7</td> <td>0</td> </tr> <tr> <td>0 0 0 0 1 0 0 0</td> <td>i i i 1 H H H H</td> <td></td> <td></td> </tr> <tr> <td>1 0 1 i i i i i</td> <td>i i i i i i i i</td> <td></td> <td></td> </tr> </table>	15	8	7	0	0 0 0 0 1 0 0 0	i i i 1 H H H H			1 0 1 i i i i i	i i i i i i i i		
15	8	7	0													
0 0 0 0 1 0 0 0	i i i 1 H H H H															
1 0 1 i i i i i	i i i i i i i i															
BMCLR #u16,DR.L	2	2	3	<table border="1"> <tr> <td>15</td> <td>8</td> <td>7</td> <td>0</td> </tr> <tr> <td>0 0 0 0 1 0 0 0</td> <td>i i i 0 H H H H</td> <td></td> <td></td> </tr> <tr> <td>1 0 1 i i i i i</td> <td>i i i i i i i i</td> <td></td> <td></td> </tr> </table>	15	8	7	0	0 0 0 0 1 0 0 0	i i i 0 H H H H			1 0 1 i i i i i	i i i i i i i i		
15	8	7	0													
0 0 0 0 1 0 0 0	i i i 0 H H H H															
1 0 1 i i i i i	i i i i i i i i															

Instruction Fields

C1	CCC	Control Registers					
000	EMR	010	-	100	—	110	—
001	VBA	011	-	101	SR	111	MCTL

DR	HHHH	Data/Address Register					
0000	D0	0100	D4	1000	R0	1100	R4
0001	D1	0101	D5	1001	R1	1101	R5
0010	D2	0110	D6	1010	R2	1110	R6
0011	D3	0111	D7	1011	R3	1111	R7

Note: If registers D8–D15 or R8–R15 are accessed instead of D0–D7 or R0–R7, a prefix is used.



#u16

iiiiiiiiiiiiii 16-bit unsigned immediate data

Operation

$0 \rightarrow (SP-u5)_i$
(i denotes bits=1 in #u16)

$0 \rightarrow (SP+s16)_i$

$0 \rightarrow (Rn)_i$

$0 \rightarrow (a16)_i$

Assembler Syntax

`BMCLR.W #u16,(SP-u5){ $0 \leq u16 < 2^{16}$ { $0 \leq u5 < 64,W$ }`

`BMCLR.W #u16,(SP+s16){ $0 \leq u16 < 2^{16}$ }{ $-2^{15} \leq s16 < 2^{15},W$ }`

`BMCLR.W #u16,(Rn){ $0 \leq u16 < 2^{16}$ }`

`BMCLR.W #u16,(a16){ $0 \leq u16 < 2^{16}$ }{ $0 \leq a16 < 2^{16},W$ }`

Description

These operations use an unsigned 16-bit immediate data mask to clear selected bits in the destination operand. For each bit i that is set (selected) in the mask, the bit i in the corresponding destination operand's bit position is cleared. These operations read from memory, modify the retrieved value, and write the new value back to that memory address, resulting in two memory accesses. The absolute addresses, offsets, and address register values must be word-aligned.

BMCLR.W #u16,(SP-u5)

Clears selected bits in the contents of a memory address pointed to by the active stack pointer (SP) with an unsigned 5-bit offset.

BMCLR.W #u16,(SP+s16)

Clears selected bits in the contents of a memory address pointed to by the active stack pointer (SP) with a 16-bit signed offset.

BMCLR.W #u16,(Rn)

Selected bits in the contents of a memory address pointed to by an address register (Rn).

BMCLR.W #u16,(a16)

Clears selected bits in the contents of a memory address pointed to by an absolute 16-bit address.

Status and Conditions that Affect Instruction

Register Address	Bit Name	Description
SR[18]	EXP	Determines which stack pointer is used when the stack pointer is an operand. Otherwise, the instruction is not affected by SR.

Status and Conditions Changed by Instruction

None.



Instruction Formats and Opcodes

Instruction	Words	Cycles	Type	Opcode																
BMCLR.W #u16, (SP-u5)	2	3	3	<table border="1"> <tr> <td>15</td> <td>8</td> <td>7</td> <td>0</td> </tr> <tr> <td>0 0 0 0 0 0 0 0</td> <td>i i i A A A A A</td> <td></td> <td></td> </tr> <tr> <td>1 0 1 i i i i i i</td> <td>i i i i i i i i</td> <td></td> <td></td> </tr> </table>	15	8	7	0	0 0 0 0 0 0 0 0	i i i A A A A A			1 0 1 i i i i i i	i i i i i i i i						
15	8	7	0																	
0 0 0 0 0 0 0 0	i i i A A A A A																			
1 0 1 i i i i i i	i i i i i i i i																			
BMCLR.W #u16, (SP+s16)	3	3	3	<table border="1"> <tr> <td>15</td> <td>8</td> <td>7</td> <td>0</td> </tr> <tr> <td>0 0 1 1 1 0 0 0</td> <td>A A A i i 0 1 1</td> <td></td> <td></td> </tr> <tr> <td>0 0 1 A A A A A</td> <td>A A A A A A A A</td> <td></td> <td></td> </tr> <tr> <td>1 0 i i i i i i i</td> <td>i i i i i i i i</td> <td></td> <td></td> </tr> </table>	15	8	7	0	0 0 1 1 1 0 0 0	A A A i i 0 1 1			0 0 1 A A A A A	A A A A A A A A			1 0 i i i i i i i	i i i i i i i i		
15	8	7	0																	
0 0 1 1 1 0 0 0	A A A i i 0 1 1																			
0 0 1 A A A A A	A A A A A A A A																			
1 0 i i i i i i i	i i i i i i i i																			
BMCLR.W #u16, (Rn)	2	2	3	<table border="1"> <tr> <td>15</td> <td>8</td> <td>7</td> <td>0</td> </tr> <tr> <td>0 0 0 1 0 0 0 0</td> <td>i i i 0 1 R R R</td> <td></td> <td></td> </tr> <tr> <td>1 0 1 i i i i i i</td> <td>i i i i i i i i</td> <td></td> <td></td> </tr> </table>	15	8	7	0	0 0 0 1 0 0 0 0	i i i 0 1 R R R			1 0 1 i i i i i i	i i i i i i i i						
15	8	7	0																	
0 0 0 1 0 0 0 0	i i i 0 1 R R R																			
1 0 1 i i i i i i	i i i i i i i i																			
BMCLR.W #u16, (a16)	3	2	3	<table border="1"> <tr> <td>15</td> <td>8</td> <td>7</td> <td>0</td> </tr> <tr> <td>0 0 1 1 1 0 0 0</td> <td>A A A i i 0 0 1</td> <td></td> <td></td> </tr> <tr> <td>0 0 1 A A A A A</td> <td>A A A A A A A A</td> <td></td> <td></td> </tr> <tr> <td>1 0 i i i i i i i</td> <td>i i i i i i i i</td> <td></td> <td></td> </tr> </table>	15	8	7	0	0 0 1 1 1 0 0 0	A A A i i 0 0 1			0 0 1 A A A A A	A A A A A A A A			1 0 i i i i i i i	i i i i i i i i		
15	8	7	0																	
0 0 1 1 1 0 0 0	A A A i i 0 0 1																			
0 0 1 A A A A A	A A A A A A A A																			
1 0 i i i i i i i	i i i i i i i i																			

Instruction Fields

Rn	RRR	Address Register					
	000 R0	010 R2	100 R4	110 R6			
	001 R1	011 R3	101 R5	111 R7			

Note: This instruction can specify R8-R15 as operands by using a high register prefix.

- a16** AAAAAAAAAAAAAAAAAA 16-bit unsigned absolute address
- #u16** iiii iiiiii 16-bit unsigned immediate data
- u5** AAAAA0 5-bit unsigned SP address offset
- s16** AAAAAAAAAAAAAAAAAA 16-bit signed SP address offset

BMSET Bit-Masked Set a 16-Bit Operand (BMU) BMSET

Operation

$1 \rightarrow C1.H_i$ (i denotes bits=1 in #u16)

$1 \rightarrow C1.L_i$ (selected bits)

$1 \rightarrow DR.H_i$ (selected bits)

$1 \rightarrow DR.L_i$ (selected bits)

Assembler Syntax

`BMSET #u16,C1.H {0 < u16 < 216}`

`BMSET #u16,C1.L {0 < u16 < 216}`

`BMSET #u16,DR.H {0 < u16 < 216}`

`BMSET #u16,DR.L {0 < u16 < 216}`

Description

These operations use an unsigned 16-bit immediate data mask to set selected bits in the destination operand. For each bit i that is set (selected) in the mask, the bit i in the corresponding destination operand's bit position is set. Bits that are not selected as well as the other part of the register are unaffected. These operations read from the register, modify the retrieved value, and write the new value back to that register.

BMSET #u16,C1.H

Sets selected bits in the HP contents of a control register (C1).

BMSET #u16,C1.L

Sets selected bits in the LP contents of a control register (C1).

BMSET #u16,DR.H

Sets selected bits in the HP contents of a data or address register (DR).

BMSET #u16,DR.L

Sets selected bits in the LP contents of a data or address register (DR).

Status and Conditions that Affect Instruction

Register Address	Bit Name	Description
SR[18]	EXP	Determines execution working mode for instructions that have these registers as an operand.

Status and Conditions Changed by Instruction

Register Address	Bit Name	Description
Ln	L	Clears the Ln bit in the destination register.

Example



iset #2436,d1.l

Register/Memory Address	Before	After
	\$2436	
L1:D1	\$0:\$0043A1243C	\$0:\$0043A1243E

Instruction Formats and Opcodes

Instruction	Words	Cycles	Type	Opcode																																					
BMSET #u16,C1.H	2	2	3	<table border="1" style="width: 100%; text-align: center;"> <tr> <td style="width: 15%;">15</td> <td style="width: 40%;"></td> <td style="width: 15%;">8</td> <td style="width: 15%;">7</td> <td style="width: 15%;">0</td> </tr> <tr> <td>0</td><td>0</td><td>0</td><td>1</td><td>0</td><td>0</td><td>0</td><td>1</td> <td>i</td><td>i</td><td>i</td><td>1</td><td>0</td><td>C</td><td>C</td><td>C</td> </tr> <tr> <td>1</td><td>0</td><td>1</td><td>i</td><td>i</td><td>i</td><td>i</td><td>i</td> <td>i</td><td>i</td><td>i</td><td>i</td><td>i</td><td>i</td><td>i</td><td>i</td> </tr> </table>	15		8	7	0	0	0	0	1	0	0	0	1	i	i	i	1	0	C	C	C	1	0	1	i	i	i	i	i	i	i	i	i	i	i	i	i
15		8	7	0																																					
0	0	0	1	0	0	0	1	i	i	i	1	0	C	C	C																										
1	0	1	i	i	i	i	i	i	i	i	i	i	i	i	i																										
BMSET #u16,C1.L	2	2	3	<table border="1" style="width: 100%; text-align: center;"> <tr> <td style="width: 15%;">15</td> <td style="width: 40%;"></td> <td style="width: 15%;">8</td> <td style="width: 15%;">7</td> <td style="width: 15%;">0</td> </tr> <tr> <td>0</td><td>0</td><td>0</td><td>1</td><td>0</td><td>0</td><td>0</td><td>1</td> <td>i</td><td>i</td><td>i</td><td>0</td><td>0</td><td>C</td><td>C</td><td>C</td> </tr> <tr> <td>1</td><td>0</td><td>1</td><td>i</td><td>i</td><td>i</td><td>i</td><td>i</td> <td>i</td><td>i</td><td>i</td><td>i</td><td>i</td><td>i</td><td>i</td><td>i</td> </tr> </table>	15		8	7	0	0	0	0	1	0	0	0	1	i	i	i	0	0	C	C	C	1	0	1	i	i	i	i	i	i	i	i	i	i	i	i	i
15		8	7	0																																					
0	0	0	1	0	0	0	1	i	i	i	0	0	C	C	C																										
1	0	1	i	i	i	i	i	i	i	i	i	i	i	i	i																										
BMSET #u16,DR.H	2	2	3	<table border="1" style="width: 100%; text-align: center;"> <tr> <td style="width: 15%;">15</td> <td style="width: 40%;"></td> <td style="width: 15%;">8</td> <td style="width: 15%;">7</td> <td style="width: 15%;">0</td> </tr> <tr> <td>0</td><td>0</td><td>0</td><td>0</td><td>1</td><td>0</td><td>0</td><td>1</td> <td>i</td><td>i</td><td>i</td><td>1</td><td>H</td><td>H</td><td>H</td><td>H</td> </tr> <tr> <td>1</td><td>0</td><td>1</td><td>i</td><td>i</td><td>i</td><td>i</td><td>i</td> <td>i</td><td>i</td><td>i</td><td>i</td><td>i</td><td>i</td><td>i</td><td>i</td> </tr> </table>	15		8	7	0	0	0	0	0	1	0	0	1	i	i	i	1	H	H	H	H	1	0	1	i	i	i	i	i	i	i	i	i	i	i	i	i
15		8	7	0																																					
0	0	0	0	1	0	0	1	i	i	i	1	H	H	H	H																										
1	0	1	i	i	i	i	i	i	i	i	i	i	i	i	i																										
BMSET #u16,DR.L	2	2	3	<table border="1" style="width: 100%; text-align: center;"> <tr> <td style="width: 15%;">15</td> <td style="width: 40%;"></td> <td style="width: 15%;">8</td> <td style="width: 15%;">7</td> <td style="width: 15%;">0</td> </tr> <tr> <td>0</td><td>0</td><td>0</td><td>0</td><td>1</td><td>0</td><td>0</td><td>1</td> <td>i</td><td>i</td><td>i</td><td>0</td><td>H</td><td>H</td><td>H</td><td>H</td> </tr> <tr> <td>1</td><td>0</td><td>1</td><td>i</td><td>i</td><td>i</td><td>i</td><td>i</td> <td>i</td><td>i</td><td>i</td><td>i</td><td>i</td><td>i</td><td>i</td><td>i</td> </tr> </table>	15		8	7	0	0	0	0	0	1	0	0	1	i	i	i	0	H	H	H	H	1	0	1	i	i	i	i	i	i	i	i	i	i	i	i	i
15		8	7	0																																					
0	0	0	0	1	0	0	1	i	i	i	0	H	H	H	H																										
1	0	1	i	i	i	i	i	i	i	i	i	i	i	i	i																										

Instruction Fields

C1 **CCC** **Control Registers**

000	EMR	010	-	100	—	110	—
001	VBA	011	-	101	SR	111	MCTL

DR **HHHH** **Data/Address Register**

0000	D0	0100	D4	1000	R0	1100	R4
0001	D1	0101	D5	1001	R1	1101	R5
0010	D2	0110	D6	1010	R2	1110	R6
0011	D3	0111	D7	1011	R3	1111	R7

Note: If registers D8–D15 or R8–R15 are accessed instead of D0–D7 or R0–R7, a prefix is used.

#u16 **iiiiiiiiiiiiiii** 16-bit unsigned immediate data

Operation

Assembler Syntax

$l \rightarrow (SP-u5)_i$ (i denotes bits=1 in #u16) $BMSET.W \#u16, (SP-u5) \{0 \leq u16 < 2^{16}\} \{0 \leq u5 < 64, W\}$

$l \rightarrow (SP+s16)_i$ (selected bits) $BMSET.W \#u16, (SP+s16) \{0 \leq u16 < 2^{16}\} \{-2^{15} \leq s16 < 2^{15}, W\}$

$l \rightarrow (Rn)_i$ (selected bits) $BMSET.W \#u16, (Rn) \{0 \leq u16 < 2^{16}\}$

$l \rightarrow (a16)_i$ (selected bits) $BMSET.W \#u16, (a16) \{0 \leq u16 < 2^{16}\} \{0 \leq a16 < 2^{16}, W\}$

Description

These operations use an unsigned 16-bit immediate data mask to set selected bits in the destination operand. For each bit i that is set (selected) in the mask, the bit i in the corresponding destination operand's bit position is set. These operations read from memory, modify the retrieved value, and write the new value back to that memory address, resulting in two memory accesses. The absolute addresses, offsets, and address register values must be word-aligned.

BMSET.W #u16,(SP-u5)

Sets selected bits in the contents of a memory address pointed to by the active stack pointer (SP) with an unsigned 5-bit offset.

BMSET.W #u16,(SP+s16)

Sets selected bits in the contents of a memory address pointed to by the active stack pointer (SP) with a 16-bit signed offset.

BMSET.W #u16,(Rn)

Sets selected bits in the contents of a memory address pointed to by an address register (Rn).

BMSET.W #u16,(a16)

Sets selected bits in the contents of a memory address pointed to by an absolute 16-bit address.

Status and Conditions that Affect Instruction

Register Address	Bit Name	Description
SR[18]	EXP	Determines which stack pointer is used when the stack pointer is an operand. Otherwise, the instruction is not affected by SR.

Status and Conditions Changed by Instruction

None.



xample

bmset.w #f111,<\$800c

Register/Memory Address	Before	After
immediate	\$F111	
(\$800C)	\$C642	\$F753

Instruction Formats and Opcodes

Instruction	Words	Cycles	Type	Opcode																
BMSET.W #u16, (SP-u5)	2	3	3	<table border="1"> <tr> <td>15</td> <td>8</td> <td>7</td> <td>0</td> </tr> <tr> <td>0 0 0 0 0 0 0 1</td> <td>i i i</td> <td>A A A A</td> <td>A</td> </tr> <tr> <td>1 0 1</td> <td>i i i i i i</td> <td>i i i i i i</td> <td>i</td> </tr> </table>	15	8	7	0	0 0 0 0 0 0 0 1	i i i	A A A A	A	1 0 1	i i i i i i	i i i i i i	i				
15	8	7	0																	
0 0 0 0 0 0 0 1	i i i	A A A A	A																	
1 0 1	i i i i i i	i i i i i i	i																	
BMSET.W #u16, (SP+s16)	3	3	3	<table border="1"> <tr> <td>15</td> <td>8</td> <td>7</td> <td>0</td> </tr> <tr> <td>0 0 1 1 1 0 0 1</td> <td>A A A</td> <td>i i</td> <td>0 1 1</td> </tr> <tr> <td>0 0 1</td> <td>A A A A A</td> <td>A A A A A A A</td> <td>A A</td> </tr> <tr> <td>1 0</td> <td>i i i i i i i</td> <td>i i i i i i i</td> <td>i</td> </tr> </table>	15	8	7	0	0 0 1 1 1 0 0 1	A A A	i i	0 1 1	0 0 1	A A A A A	A A A A A A A	A A	1 0	i i i i i i i	i i i i i i i	i
15	8	7	0																	
0 0 1 1 1 0 0 1	A A A	i i	0 1 1																	
0 0 1	A A A A A	A A A A A A A	A A																	
1 0	i i i i i i i	i i i i i i i	i																	
BMSET.W #u16, (Rn)	2	2	3	<table border="1"> <tr> <td>15</td> <td>8</td> <td>7</td> <td>0</td> </tr> <tr> <td>0 0 0 1 0 0 0 1</td> <td>i i i</td> <td>0 1</td> <td>R R R</td> </tr> <tr> <td>1 0 1</td> <td>i i i i i i</td> <td>i i i i i i</td> <td>i</td> </tr> </table>	15	8	7	0	0 0 0 1 0 0 0 1	i i i	0 1	R R R	1 0 1	i i i i i i	i i i i i i	i				
15	8	7	0																	
0 0 0 1 0 0 0 1	i i i	0 1	R R R																	
1 0 1	i i i i i i	i i i i i i	i																	
BMSET.W #u16, (a16)	3	2	3	<table border="1"> <tr> <td>15</td> <td>8</td> <td>7</td> <td>0</td> </tr> <tr> <td>0 0 1 1 1 0 0 1</td> <td>A A A</td> <td>i i</td> <td>0 0 1</td> </tr> <tr> <td>0 0 1</td> <td>A A A A A</td> <td>A A A A A A A</td> <td>A A</td> </tr> <tr> <td>1 0</td> <td>i i i i i i i</td> <td>i i i i i i i</td> <td>i</td> </tr> </table>	15	8	7	0	0 0 1 1 1 0 0 1	A A A	i i	0 0 1	0 0 1	A A A A A	A A A A A A A	A A	1 0	i i i i i i i	i i i i i i i	i
15	8	7	0																	
0 0 1 1 1 0 0 1	A A A	i i	0 0 1																	
0 0 1	A A A A A	A A A A A A A	A A																	
1 0	i i i i i i i	i i i i i i i	i																	

Instruction Fields

Rn	RRR	Address Register						
	000	R0	010	R2	100	R4	110	R6
	001	R1	011	R3	101	R5	111	R7

Note: This instruction can specify R8-R15 as operands by using a high register prefix.

a16	AAAAAAAAAAAAAAAA	16-bit unsigned absolute address
#u16	iiiiiiiiiiiiiiii	16-bit unsigned immediate data
u5	AAAAA0	5-bit unsigned SP address offset
s16	AAAAAAAAAAAAAAAA	16-bit signed SP address offset



BMTSET

Bit-Masked Test and Set a 16-Bit Operand (BMU)

BMTSET

Operation

$1 \rightarrow DR.H_i$ (i denotes bits=1 in #u16)
if (all selected bits were set), then $1 \rightarrow T$, else $0 \rightarrow T$

$1 \rightarrow DR.L_i$ (selected bits)
if (all selected bits were set), then $1 \rightarrow T$, else $0 \rightarrow T$

Assembler Syntax

`BMTSET #u16,DR.H {0 ≤ u16 < 216}`

`BMTSET #u16,DR.L {0 ≤ u16 < 216}`

Description

These operations use an unsigned 16-bit immediate data mask to test and set selected bits in the destination operand. For each bit i that is set (selected) in the mask, the bit i in the corresponding destination operand's bit position is set. Unselected bits are unaffected. If all selected bits were set when the data was read, the T bit is set. If at least one of the selected bits was not set, the T bit is cleared. This operation reads from a register, modifies the retrieved value, and writes the new value back to that register.

BMTSET #u16,DR.H

Tests and sets selected bits in the HP contents of a data or address register (DR).

BMTSET #u16,DR.L

Tests and sets selected bits in the LP contents of a data or address register (DR).

Status and Conditions that Affect Instruction

None.

Status and Conditions Changed by Instruction

Register Address	Bit Name	Description
SR[1]	T	Set if all the bits selected by the mask were set, cleared otherwise.
Ln	L	Clears the Ln bit in the destination data register.

Example 1

```
bmtset #111f,d1.l
```

Register/Memory Address	Before	After
SR	\$00E4 0000	\$00E4 0000
immediate	\$111F	
d1	\$00 1234 5678	\$00 1234 577F



Example 2

```
bmtset #4238,d4.l
```

Register/Memory Address	Before	After
SR	\$00E4 0000	\$00E4 0002
immediate	\$4238	
d4	\$00 1234 5678	\$00 1234 5678

Instruction Formats and Opcodes

Instruction	Words	Cycles	Type	Opcode												
BMTSET #u16,DR.H	2	2	3	<table border="1"> <tr> <td>15</td> <td>8</td> <td>7</td> <td>0</td> </tr> <tr> <td>0 0 0 0 1 1 1 0</td> <td>i i i 1 H H H H</td> <td></td> <td></td> </tr> <tr> <td>1 0 1 i i i i i</td> <td>i i i i i i i i</td> <td></td> <td></td> </tr> </table>	15	8	7	0	0 0 0 0 1 1 1 0	i i i 1 H H H H			1 0 1 i i i i i	i i i i i i i i		
15	8	7	0													
0 0 0 0 1 1 1 0	i i i 1 H H H H															
1 0 1 i i i i i	i i i i i i i i															
BMTSET #u16,DR.L	2	2	3	<table border="1"> <tr> <td>15</td> <td>8</td> <td>7</td> <td>0</td> </tr> <tr> <td>0 0 0 0 1 1 1 0</td> <td>i i i 0 H H H H</td> <td></td> <td></td> </tr> <tr> <td>1 0 1 i i i i i</td> <td>i i i i i i i i</td> <td></td> <td></td> </tr> </table>	15	8	7	0	0 0 0 0 1 1 1 0	i i i 0 H H H H			1 0 1 i i i i i	i i i i i i i i		
15	8	7	0													
0 0 0 0 1 1 1 0	i i i 0 H H H H															
1 0 1 i i i i i	i i i i i i i i															

Instruction Fields

DR	HHHH	Data/Address Register					
0000	D0	0100	D4	1000	R0	1100	R4
0001	D1	0101	D5	1001	R1	1101	R5
0010	D2	0110	D6	1010	R2	1110	R6
0011	D3	0111	D7	1011	R3	1111	R7

Note: This instruction can specify D8-D15 or R8-R15 as operands by using a high register prefix.

#u16 `iiiiiiiiiiiiiiii` 16-bit unsigned immediate data



BMTSET.W Bit-Masked Test and Set a 16-Bit Operand in Memory (BMU) BMTSET.W

Operation

$1 \rightarrow (SP-u5)_i$ (i denotes bits=1 in #u16)
if (all selected bits were set), then $1 \rightarrow T$, else $0 \rightarrow T$

$1 \rightarrow (SP+s16)_i$
if (all selected bits were set), then $1 \rightarrow T$, else $0 \rightarrow T$

$1 \rightarrow (Rn)_i$
if (all selected bits were set), then $1 \rightarrow T$, else $0 \rightarrow T$

$1 \rightarrow (a16)_i$
if (all selected bits were set), then $1 \rightarrow T$, else $0 \rightarrow T$

Assembler Syntax

`BMTSET.W #u16, (SP-u5) { $0 \leq u16 < 2^{16}$ }
{ $0 \leq u5 < 64, W$ }`

`BMTSET.W #u16, (SP+s16) { $0 \leq u16 < 2^{16}$ }
{ $-2^{15} \leq s16 < 2^{15}, W$ }`

`BMTSET.W #u16, (Rn) { $0 \leq u16 < 2^{16}$ }`

`BMTSET.W #u16, (a16) { $0 \leq u16 < 2^{16}$ }
{ $0 \leq a16 < 2^{16}, W$ }`

Description

These operations use an unsigned 16-bit immediate data mask to test and set selected bits in the destination operand. For each bit i that is set (selected) in the mask, the bit i in the destination operand's corresponding bit position is set. Unselected bits are unaffected. These operations read from memory, modify the retrieved value, and attempt to write the new value back to that memory address. These operations result in two memory accesses.

This instruction is intended for semaphore support in a multi-process shared memory environment. Typically, the process that wants to get exclusive control of a semaphore tries to set bits in the memory using this instruction. This action can fail if all of the bits are already set. It can also fail in case of protection violation, or if another process has locked the bus or written to the same memory address between the read and write cycles of this instruction. It is the responsibility of the memory system to inform the core of failures due to the latter case. Both failures cause the T bit to be set. The process attempting to set the semaphore should test the T bit after the instruction is executed in order to determine if the semaphore is set or not. The absolute addresses, offsets, and address register values must be word-aligned.

Although this instruction is designed with semaphores in mind, it can be used for other applications.

BMTSET.W #u16,(SP-u5)

Tests and sets selected bits in the contents of a memory address pointed to by the active stack pointer (SP) with an unsigned 5-bit offset.

BMTSET.W #u16,(SP+s16)

Tests and sets selected bits in the contents of a memory address pointed to by the active stack pointer (SP) with a 16-bit signed offset.

BMTSET.W #u16,(Rn)

Tests and sets selected bits in the contents of a memory address pointed to by an address register (Rn).

BMTSET.W #u16,(a16)

Tests and sets selected bits in the contents of a memory address pointed to by an absolute 16-bit address.

Status and Conditions that Affect Instruction

Register Address	Bit Name	Description
SR[18]	EXP	Determines which stack pointer is used when the stack pointer is an operand. Otherwise, the instruction is not affected by SR.

Status and Conditions Changed by Instruction

Register Address	Bit Name	Description
SR[1]	T	Set if all the bits selected by the mask are set, or the memory access fails; cleared otherwise.

Example

```
bmtset.w #4328, ($c)
```

Register/Memory Address	Before	After
immediate	\$4238	
(\$C)	\$5678	\$5678
SR	\$00E4 0000	\$00E4 0002

Instruction Formats and Opcodes

Instruction	Words	Cycles	Type	Opcode																
BMTSET.W #u16,(SP-u5)	2	3	3	<table border="1"> <tr> <td>15</td> <td>8</td> <td>7</td> <td>0</td> </tr> <tr> <td>0 0 0 0 0 1 1 0</td> <td>i i i</td> <td>A A A</td> <td>A A A</td> </tr> <tr> <td>1 0 1 i i i i i</td> <td>i i i</td> <td>i i i</td> <td>i i i</td> </tr> </table>	15	8	7	0	0 0 0 0 0 1 1 0	i i i	A A A	A A A	1 0 1 i i i i i	i i i	i i i	i i i				
15	8	7	0																	
0 0 0 0 0 1 1 0	i i i	A A A	A A A																	
1 0 1 i i i i i	i i i	i i i	i i i																	
BMTSET.W #u16,(SP+s16)	3	3	3	<table border="1"> <tr> <td>15</td> <td>8</td> <td>7</td> <td>0</td> </tr> <tr> <td>0 0 1 1 1 1 1 0</td> <td>A A A</td> <td>i i</td> <td>0 1 1</td> </tr> <tr> <td>0 0 1 A A A A A</td> <td>A A A</td> <td>A A A</td> <td>A A A</td> </tr> <tr> <td>1 0 i i i i i i</td> <td>i i i</td> <td>i i i</td> <td>i i i</td> </tr> </table>	15	8	7	0	0 0 1 1 1 1 1 0	A A A	i i	0 1 1	0 0 1 A A A A A	A A A	A A A	A A A	1 0 i i i i i i	i i i	i i i	i i i
15	8	7	0																	
0 0 1 1 1 1 1 0	A A A	i i	0 1 1																	
0 0 1 A A A A A	A A A	A A A	A A A																	
1 0 i i i i i i	i i i	i i i	i i i																	
BMTSET.W #u16,(Rn)	2	2	3	<table border="1"> <tr> <td>15</td> <td>8</td> <td>7</td> <td>0</td> </tr> <tr> <td>0 0 0 1 0 1 1 0</td> <td>i i i</td> <td>0 1</td> <td>R R R</td> </tr> <tr> <td>1 0 1 i i i i i</td> <td>i i i</td> <td>i i i</td> <td>i i i</td> </tr> </table>	15	8	7	0	0 0 0 1 0 1 1 0	i i i	0 1	R R R	1 0 1 i i i i i	i i i	i i i	i i i				
15	8	7	0																	
0 0 0 1 0 1 1 0	i i i	0 1	R R R																	
1 0 1 i i i i i	i i i	i i i	i i i																	
BMTSET.W #u16,(a16)	3	2	3	<table border="1"> <tr> <td>15</td> <td>8</td> <td>7</td> <td>0</td> </tr> <tr> <td>0 0 1 1 1 1 1 0</td> <td>A A A</td> <td>i i</td> <td>0 0 1</td> </tr> <tr> <td>0 0 1 A A A A A</td> <td>A A A</td> <td>A A A</td> <td>A A A</td> </tr> <tr> <td>1 0 i i i i i i</td> <td>i i i</td> <td>i i i</td> <td>i i i</td> </tr> </table>	15	8	7	0	0 0 1 1 1 1 1 0	A A A	i i	0 0 1	0 0 1 A A A A A	A A A	A A A	A A A	1 0 i i i i i i	i i i	i i i	i i i
15	8	7	0																	
0 0 1 1 1 1 1 0	A A A	i i	0 0 1																	
0 0 1 A A A A A	A A A	A A A	A A A																	
1 0 i i i i i i	i i i	i i i	i i i																	

Instruction Fields

Rn	RRR	Address Register					
	000 R0	010 R2	100 R4	110 R6			
	001 R1	011 R3	101 R5	111 R7			

Note: This instruction can specify R8-R15 as operands by using a high register prefix.

a16	AAAAAAAAAAAAAAAA	16-bit unsigned absolute address
#u16	iiiiiiiiiiiiiiii	16-bit unsigned immediate data
u5	AAAAA0	5-bit unsigned SP address offset
s16	AAAAAAAAAAAAAAAA	16-bit signed SP address offset



Operation

Assembler Syntax

if ($\#u16 \& C1.H$) == \$0000, then 1 \rightarrow T, else 0 \rightarrow T `BMTSTC #u16,C1.H {0 ≤ u16 < 216}`

if ($\#u16 \& C1.L$) == \$0000, then 1 \rightarrow T, else 0 \rightarrow T `BMTSTC #u16,C1.L {0 ≤ u16 < 216}`

if ($\#u16 \& DR.H$) == \$0000, then 1 \rightarrow T, else 0 \rightarrow T `BMTSTC #u16,DR.H {0 ≤ u16 < 216}`

if ($\#u16 \& DR.L$) == \$0000, then 1 \rightarrow T, else 0 \rightarrow T `BMTSTC #u16,DR.L {0 ≤ u16 < 216}`

Description

These operations use an unsigned 16-bit immediate data mask to determine if all selected bits in an operand are cleared. If all the selected bits are cleared, the T bit is set; if not, the T bit is cleared.

BMTSTC #u16,C1.H

Tests selected bits in the HP contents of a control register (C1).

BMTSTC #u16,C1.L

Tests selected bits in the LP contents of a control register (C1).

BMTSTC #u16,DR.H

Tests selected bits in the HP contents of a data or address register (DR).

BMTSTC #u16,DR.L

Tests selected bits in the LP contents of a data or address register (DR).

Status and Conditions that Affect Instruction

Register Address	Bit Name	Description
SR[18]	EXP	Determines working mode, and which SR or EMR is used for instructions that have these registers as an operand.

Status and Conditions Changed by Instruction

Register Address	Bit Name	Description
SR[1]	T	Set if all the bits selected by the mask are clear, cleared otherwise.

Example

```
bmtstc #$8a59,d7.h
```

Register/Memory Address	Before	After
immediate		\$8A590000

Register/Memory Address	Before	After
L7:D7	\$0:\$0024A60000	
SR	\$00E40000	\$00E40002

Instruction Formats and Opcodes

Instruction	Words	Cycles	Type	Opcode												
BMTSTC #u16,C1.H	2	2	3	<table border="1"> <tr> <td>15</td> <td>8</td> <td>7</td> <td>0</td> </tr> <tr> <td>0 0 0 1 0 1 0 0</td> <td>i i i 1 0</td> <td>C C C</td> <td></td> </tr> <tr> <td>1 0 1 i i i i i</td> <td>i i i i i i i i</td> <td></td> <td></td> </tr> </table>	15	8	7	0	0 0 0 1 0 1 0 0	i i i 1 0	C C C		1 0 1 i i i i i	i i i i i i i i		
15	8	7	0													
0 0 0 1 0 1 0 0	i i i 1 0	C C C														
1 0 1 i i i i i	i i i i i i i i															
BMTSTC #u16,C1.L	2	2	3	<table border="1"> <tr> <td>15</td> <td>8</td> <td>7</td> <td>0</td> </tr> <tr> <td>0 0 0 1 0 1 0 0</td> <td>i i i 0 0</td> <td>C C C</td> <td></td> </tr> <tr> <td>1 0 1 i i i i i</td> <td>i i i i i i i i</td> <td></td> <td></td> </tr> </table>	15	8	7	0	0 0 0 1 0 1 0 0	i i i 0 0	C C C		1 0 1 i i i i i	i i i i i i i i		
15	8	7	0													
0 0 0 1 0 1 0 0	i i i 0 0	C C C														
1 0 1 i i i i i	i i i i i i i i															
BMTSTC #u16,DR.H	2	2	3	<table border="1"> <tr> <td>15</td> <td>8</td> <td>7</td> <td>0</td> </tr> <tr> <td>0 0 0 0 1 1 0 0</td> <td>i i i 1</td> <td>H H H H</td> <td></td> </tr> <tr> <td>1 0 1 i i i i i</td> <td>i i i i i i i i</td> <td></td> <td></td> </tr> </table>	15	8	7	0	0 0 0 0 1 1 0 0	i i i 1	H H H H		1 0 1 i i i i i	i i i i i i i i		
15	8	7	0													
0 0 0 0 1 1 0 0	i i i 1	H H H H														
1 0 1 i i i i i	i i i i i i i i															
BMTSTC #u16,DR.L	2	2	3	<table border="1"> <tr> <td>15</td> <td>8</td> <td>7</td> <td>0</td> </tr> <tr> <td>0 0 0 0 1 1 0 0</td> <td>i i i 0</td> <td>H H H H</td> <td></td> </tr> <tr> <td>1 0 1 i i i i i</td> <td>i i i i i i i i</td> <td></td> <td></td> </tr> </table>	15	8	7	0	0 0 0 0 1 1 0 0	i i i 0	H H H H		1 0 1 i i i i i	i i i i i i i i		
15	8	7	0													
0 0 0 0 1 1 0 0	i i i 0	H H H H														
1 0 1 i i i i i	i i i i i i i i															

Instruction Fields

C1	CCC	Control Registers					
000	EMR	010	-	100	—	110	—
001	VBA	011	-	101	SR	111	MCTL

DR	HHHH	Data/Address Register					
0000	D0	0100	D4	1000	R0	1100	R4
0001	D1	0101	D5	1001	R1	1101	R5
0010	D2	0110	D6	1010	R2	1110	R6
0011	D3	0111	D7	1011	R3	1111	R7

Note: If registers D8–D15 or R8–R15 are accessed instead of D0–D7 or R0–R7, a prefix is used.

#u16 **iiiiiiiiiiiiiiii** 16-bit unsigned immediate data



BMTSTC.W

Bit-Masked Test a 16-Bit Operand in Memory If Clear (BMU)

BMTSTC.W

Operation

if ($\#u16 \& (SP-u5)$) == \$0000, then 1→T else 0→T

if ($\#u16 \& (SP+s16)$) == \$0000, then 1→T else 0→T

if ($\#u16 \& (Rn)$) == \$0000, then 1→T else 0→T

if ($\#u16 \& (a16)$) == \$0000, then 1→T else 0→T

Assembler Syntax

BMTSTC.W $\#u16, (SP-u5) \{0 \leq u16 < 2^{16}\}$
 $\{0 \leq u5 < 64, W\}$

BMTSTC.W $\#u16, (SP+s16) \{0 \leq u16 < 2^{16}\}$
 $\{-2^{15} \leq s16 < 2^{15}, W\}$

BMTSTC.W $\#u16, (Rn) \{0 \leq u16 < 2^{16}\}$

BMTSTC.W $\#u16, (a16) \{0 \leq u16 < 2^{16}\}$
 $\{0 \leq a16 < 2^{16}, W\}$

Description

These operations use an unsigned 16-bit immediate data mask to determine if all selected bits in an operand are cleared. If all the selected bits are cleared, the T bit is set; if not, the T bit is cleared. The absolute addresses, offsets, and address register values must be word-aligned.

BMTSTC.W $\#u16, (SP-u5)$

Tests selected bits in the contents of a memory address pointed to by the active stack pointer (SP) with an unsigned 5-bit offset.

BMTSTC.W $\#u16, (SP+s16)$

Tests selected bits in the contents of a memory address pointed to by the active stack pointer (SP) with a signed 16-bit offset.

BMTSTC.W $\#u16, (Rn)$

Tests selected bits in the contents of a memory address pointed to by an address register (Rn).

BMTSTC.W $\#u16, (a16)$

Tests selected bits in the contents of a memory address pointed to by an absolute 16-bit address.

Status and Conditions that Affect Instruction

Register Address	Bit Name	Description
SR[18]	EXP	Determines which stack pointer is used when the stack pointer is an operand. Otherwise, the instruction is not affected by SR.

Status and Conditions Changed by Instruction

Register Address	Bit Name	Description
SR[1]	T	Set if all the bits selected by the mask are clear, cleared otherwise.

Example

```
bmtstc.w #8A59, (r0)
```

Register/Memory Address	Before	After
immediate	\$8A59	
R0	\$0000 0002	\$0000 0002
(\$0002)	\$0000 24A6	\$0000 24A6
SR	\$00E4 0000	\$00E4 0002

```

$24A6 --0010 0100 1010 0110
mask $8A59 --1000 1010 0101 1001

```

Instruction Formats and Opcodes

Instruction	Words	Cycles	Type	Opcode																
BMTSTC.W #u16,(SP-u5)	2	3	3	<table border="1"> <tr> <td>15</td> <td>8</td> <td>7</td> <td>0</td> </tr> <tr> <td>0 0 0 0 0 1 0 0</td> <td>i i i</td> <td>A A A</td> <td>A A A</td> </tr> <tr> <td>1 0 1 i i i i i</td> <td>i i i</td> <td>i i i</td> <td>i i i</td> </tr> </table>	15	8	7	0	0 0 0 0 0 1 0 0	i i i	A A A	A A A	1 0 1 i i i i i	i i i	i i i	i i i				
15	8	7	0																	
0 0 0 0 0 1 0 0	i i i	A A A	A A A																	
1 0 1 i i i i i	i i i	i i i	i i i																	
BMTSTC.W #u16,(SP+s16)	3	3	3	<table border="1"> <tr> <td>15</td> <td>8</td> <td>7</td> <td>0</td> </tr> <tr> <td>0 0 1 1 1 1 0 0</td> <td>A A A</td> <td>i i</td> <td>0 1 1</td> </tr> <tr> <td>0 0 1 A A A A A</td> <td>A A A</td> <td>A A A</td> <td>A A A</td> </tr> <tr> <td>1 0 i i i i i i</td> <td>i i i</td> <td>i i i</td> <td>i i i</td> </tr> </table>	15	8	7	0	0 0 1 1 1 1 0 0	A A A	i i	0 1 1	0 0 1 A A A A A	A A A	A A A	A A A	1 0 i i i i i i	i i i	i i i	i i i
15	8	7	0																	
0 0 1 1 1 1 0 0	A A A	i i	0 1 1																	
0 0 1 A A A A A	A A A	A A A	A A A																	
1 0 i i i i i i	i i i	i i i	i i i																	
BMTSTC.W #u16,(Rn)	2	2	3	<table border="1"> <tr> <td>15</td> <td>8</td> <td>7</td> <td>0</td> </tr> <tr> <td>0 0 0 1 0 1 0 0</td> <td>i i i</td> <td>0 1</td> <td>R R R</td> </tr> <tr> <td>1 0 1 i i i i i</td> <td>i i i</td> <td>i i i</td> <td>i i i</td> </tr> </table>	15	8	7	0	0 0 0 1 0 1 0 0	i i i	0 1	R R R	1 0 1 i i i i i	i i i	i i i	i i i				
15	8	7	0																	
0 0 0 1 0 1 0 0	i i i	0 1	R R R																	
1 0 1 i i i i i	i i i	i i i	i i i																	
BMTSTC.W #u16,(a16)	3	2	3	<table border="1"> <tr> <td>15</td> <td>8</td> <td>7</td> <td>0</td> </tr> <tr> <td>0 0 1 1 1 1 0 0</td> <td>A A A</td> <td>i i</td> <td>0 0 1</td> </tr> <tr> <td>0 0 1 A A A A A</td> <td>A A A</td> <td>A A A</td> <td>A A A</td> </tr> <tr> <td>1 0 i i i i i i</td> <td>i i i</td> <td>i i i</td> <td>i i i</td> </tr> </table>	15	8	7	0	0 0 1 1 1 1 0 0	A A A	i i	0 0 1	0 0 1 A A A A A	A A A	A A A	A A A	1 0 i i i i i i	i i i	i i i	i i i
15	8	7	0																	
0 0 1 1 1 1 0 0	A A A	i i	0 0 1																	
0 0 1 A A A A A	A A A	A A A	A A A																	
1 0 i i i i i i	i i i	i i i	i i i																	

Instruction Fields

Rn	RRR	Address Register						
	000	R0	010	R2	100	R4	110	R6
	001	R1	011	R3	101	R5	111	R7

Note: This instruction can specify R8-R15 as operands by using a high register prefix.

a16	AAAAAAAAAAAAAAAA	16-bit unsigned absolute address
#u16	iiiiiiiiiiiiiiii	16-bit unsigned immediate data
u5	AAAAA0	5-bit unsigned SP address offset
s16	AAAAAAAAAAAAAAAA	16-bit signed SP address offset

Operation

if ($\#u16 \& \sim C1.H = \$0000$), then $1 \rightarrow T$, else $0 \rightarrow T$

if ($\#u16 \& \sim C1.L = \$0000$), then $1 \rightarrow T$, else $0 \rightarrow T$

if ($\#u16 \& \sim DR.H = \$0000$), then $1 \rightarrow T$, else $0 \rightarrow T$

if ($\#u16 \& \sim DR.L = \$0000$), then $1 \rightarrow T$, else $0 \rightarrow T$

Assembler Syntax

`BMTSTS #u16,C1.H` $\{0 \leq u16 < 2^{16}\}$

`BMTSTS #u16,C1.L` $\{0 \leq u16 < 2^{16}\}$

`BMTSTS #u16,DR.H` $\{0 \leq u16 < 2^{16}\}$

`BMTSTS #u16,DR.L` $\{0 \leq u16 < 2^{16}\}$

Description

These operations use an unsigned 16-bit immediate data mask to determine if all selected bits in an operand are set. If all the selected bits are set, the T bit is set; if not, the T bit is cleared.

BMTSTS #u16,C1.H

Tests selected bits in the HP contents of a control register (C1).

BMTSTS #u16,C1.L

Tests selected bits in the LP contents of a control register (C1).

BMTSTS #u16,DR.H

Tests selected bits in the HP contents of a data or address register (DR).

BMTSTS #u16,DR.L

Tests selected bits in the LP contents of a data or address register (DR).

Status and Conditions Changed by Instruction

Register Address	Bit Name	Description
SR[1]	T	Set if all the bits selected by the mask are set, cleared otherwise.



Example

```
bmtsts #24a6,d7.h
```

Register/Memory Address	Before	After
immediate	\$24A60000	
L7:D7	\$0:\$0024A60560	
SR	\$00E40000	\$00E40002

Instruction Formats and Opcodes

Instruction	Words	Cycles	Type	Opcode												
BMTSTS #u16,C1.H	2	2	3	<table border="1"> <tr> <td>15</td> <td>8</td> <td>7</td> <td>0</td> </tr> <tr> <td>0 0 0 1 0 1 0 1</td> <td>i i i 1 0 C C C</td> <td></td> <td></td> </tr> <tr> <td>1 0 1 i i i i i</td> <td>i i i i i i i i</td> <td></td> <td></td> </tr> </table>	15	8	7	0	0 0 0 1 0 1 0 1	i i i 1 0 C C C			1 0 1 i i i i i	i i i i i i i i		
15	8	7	0													
0 0 0 1 0 1 0 1	i i i 1 0 C C C															
1 0 1 i i i i i	i i i i i i i i															
BMTSTS #u16,C1.L	2	2	3	<table border="1"> <tr> <td>15</td> <td>8</td> <td>7</td> <td>0</td> </tr> <tr> <td>0 0 0 1 0 1 0 1</td> <td>i i i 0 0 C C C</td> <td></td> <td></td> </tr> <tr> <td>1 0 1 i i i i i</td> <td>i i i i i i i i</td> <td></td> <td></td> </tr> </table>	15	8	7	0	0 0 0 1 0 1 0 1	i i i 0 0 C C C			1 0 1 i i i i i	i i i i i i i i		
15	8	7	0													
0 0 0 1 0 1 0 1	i i i 0 0 C C C															
1 0 1 i i i i i	i i i i i i i i															
BMTSTS #u16,DR.H	2	2	3	<table border="1"> <tr> <td>15</td> <td>8</td> <td>7</td> <td>0</td> </tr> <tr> <td>0 0 0 0 1 1 0 1</td> <td>i i i 1 H H H H</td> <td></td> <td></td> </tr> <tr> <td>1 0 1 i i i i i</td> <td>i i i i i i i i</td> <td></td> <td></td> </tr> </table>	15	8	7	0	0 0 0 0 1 1 0 1	i i i 1 H H H H			1 0 1 i i i i i	i i i i i i i i		
15	8	7	0													
0 0 0 0 1 1 0 1	i i i 1 H H H H															
1 0 1 i i i i i	i i i i i i i i															
BMTSTS #u16,DR.L	2	2	3	<table border="1"> <tr> <td>15</td> <td>8</td> <td>7</td> <td>0</td> </tr> <tr> <td>0 0 0 0 1 1 0 1</td> <td>i i i 0 H H H H</td> <td></td> <td></td> </tr> <tr> <td>1 0 1 i i i i i</td> <td>i i i i i i i i</td> <td></td> <td></td> </tr> </table>	15	8	7	0	0 0 0 0 1 1 0 1	i i i 0 H H H H			1 0 1 i i i i i	i i i i i i i i		
15	8	7	0													
0 0 0 0 1 1 0 1	i i i 0 H H H H															
1 0 1 i i i i i	i i i i i i i i															

Instruction Fields

C1	CCC	Control Registers					
000	EMR	010	-	100	—	110	—
001	VBA	011	-	101	SR	111	MCTL

DR	HHHH	Data/Address Register					
0000	D0	0100	D4	1000	R0	1100	R4
0001	D1	0101	D5	1001	R1	1101	R5
0010	D2	0110	D6	1010	R2	1110	R6
0011	D3	0111	D7	1011	R3	1111	R7

Note: If registers D8–D15 or R8–R15 are accessed instead of D0–D7 or R0–R7, a prefix is used.

#u16 iiii...iiiiii 16-bit unsigned immediate data



BMTSTS.W

Bit-Masked Test a 16-Bit Operand in Memory (BMU)

BMTSTS.W

Operation

if ($\#u16 \& \sim(SP-u5) = \$0000$), then $1 \rightarrow T$,
else $0 \rightarrow T$

if ($\#u16 \& \sim(SP+s16) = \0000), then $1 \rightarrow T$,
else $0 \rightarrow T$

if ($\#u16 \& \sim(Rn) = \$0000$), then $1 \rightarrow T$, else $0 \rightarrow T$

if ($\#u16 \& \sim(a16) = \$0000$), then $1 \rightarrow T$, else $0 \rightarrow T$

Assembler Syntax

`BMTSTS.W #u16, (SP-u5) {0 ≤ u16 < 216}
{0 ≤ u5 < 64, W}`

`BMTSTS.W #u16, (SP+s16) {0 ≤ u16 < 216}
{-215 ≤ s16 < 215, W}`

`BMTSTS.W #u16, (Rn) {0 ≤ u16 < 216}`

`BMTSTS.W #u16, (a16) {0 ≤ u16 < 216}
{0 ≤ a16 < 216, W}`

Description

These operations use an unsigned 16-bit immediate data mask to determine if all selected bits in an operand are set. If all the selected bits are set, the T bit is set; if not, the T bit is cleared. The absolute addresses, offsets, and address register values must be word-aligned.

BMTSTS.W #u16,(SP-u5)

Tests selected bits in the contents of a memory address pointed to by the active stack pointer (SP) with an unsigned 5-bit offset.

BMTSTS.W #u16,(SP+s16)

Tests selected bits in the contents of a memory address pointed to by the active stack pointer (SP) with a 16-bit signed offset.

BMTSTS.W #u16,(Rn)

Tests selected bits in the contents of a memory address pointed to by an address register (Rn).

BMTSTS.W #u16,(a16)

Tests selected bits in the contents of a memory address pointed to by an absolute 16-bit address.

Status and Conditions that Affect Instruction

Register Address	Bit Name	Description
SR[18]	EXP	Determines which stack pointer is used when the stack pointer is an operand. Otherwise, the instruction is not affected by SR.

Status and Conditions Changed by Instruction

Register Address	Bit Name	Description
SR[1]	T	Set if all the selected bits in the mask are set, or the memory access fails; cleared otherwise.

Example

```
bmtsts.w #$0428, (r0)
```

Register/Memory Address	Before	After
immediate	\$0428	
(r0)	\$16FC	\$16FC
sr	\$00E4 0000	\$00E4 0002
In binary, \$0428	0000010000101000	
\$16FC	0001011011111100	

Instruction Formats and Opcodes

Instruction	Words	Cycles	Type	Opcode																
BMTSTS.W #u16, (SP-u5)	2	3	3	<table border="1"> <tr> <td>15</td> <td>8</td> <td>7</td> <td>0</td> </tr> <tr> <td>0 0 0 0 0 1 0 1</td> <td>i i i A A A A A</td> <td></td> <td></td> </tr> <tr> <td>1 0 1 i i i i i</td> <td>i i i i i i i i</td> <td></td> <td></td> </tr> </table>	15	8	7	0	0 0 0 0 0 1 0 1	i i i A A A A A			1 0 1 i i i i i	i i i i i i i i						
15	8	7	0																	
0 0 0 0 0 1 0 1	i i i A A A A A																			
1 0 1 i i i i i	i i i i i i i i																			
BMTSTS.W #u16, (SP+s16)	3	3	3	<table border="1"> <tr> <td>15</td> <td>8</td> <td>7</td> <td>0</td> </tr> <tr> <td>0 0 1 1 1 1 0 1</td> <td>A A A i i 0 1 1</td> <td></td> <td></td> </tr> <tr> <td>0 0 1 A A A A A</td> <td>A A A A A A A A</td> <td></td> <td></td> </tr> <tr> <td>1 0 i i i i i i</td> <td>i i i i i i i i</td> <td></td> <td></td> </tr> </table>	15	8	7	0	0 0 1 1 1 1 0 1	A A A i i 0 1 1			0 0 1 A A A A A	A A A A A A A A			1 0 i i i i i i	i i i i i i i i		
15	8	7	0																	
0 0 1 1 1 1 0 1	A A A i i 0 1 1																			
0 0 1 A A A A A	A A A A A A A A																			
1 0 i i i i i i	i i i i i i i i																			
BMTSTS.W #u16, (Rn)	2	2	3	<table border="1"> <tr> <td>15</td> <td>8</td> <td>7</td> <td>0</td> </tr> <tr> <td>0 0 0 1 0 1 0 1</td> <td>i i i 0 1 R R R</td> <td></td> <td></td> </tr> <tr> <td>1 0 1 i i i i i</td> <td>i i i i i i i i</td> <td></td> <td></td> </tr> </table>	15	8	7	0	0 0 0 1 0 1 0 1	i i i 0 1 R R R			1 0 1 i i i i i	i i i i i i i i						
15	8	7	0																	
0 0 0 1 0 1 0 1	i i i 0 1 R R R																			
1 0 1 i i i i i	i i i i i i i i																			
BMTSTS.W #u16, (a16)	3	2	3	<table border="1"> <tr> <td>15</td> <td>8</td> <td>7</td> <td>0</td> </tr> <tr> <td>0 0 1 1 1 1 0 1</td> <td>A A A i i 0 0 1</td> <td></td> <td></td> </tr> <tr> <td>0 0 1 A A A A A</td> <td>A A A A A A A A</td> <td></td> <td></td> </tr> <tr> <td>1 0 i i i i i i</td> <td>i i i i i i i i</td> <td></td> <td></td> </tr> </table>	15	8	7	0	0 0 1 1 1 1 0 1	A A A i i 0 0 1			0 0 1 A A A A A	A A A A A A A A			1 0 i i i i i i	i i i i i i i i		
15	8	7	0																	
0 0 1 1 1 1 0 1	A A A i i 0 0 1																			
0 0 1 A A A A A	A A A A A A A A																			
1 0 i i i i i i	i i i i i i i i																			

Instruction Fields

Rn	RRR	Address Register						
	000	R0	010	R2	100	R4	110	R6
	001	R1	011	R3	101	R5	111	R7

Note: This instruction can specify R8-R15 as operands by using a high register prefix.

a16	AAAAAAAAAAAAAAAA	16-bit unsigned absolute address
#u16	iiiiiiiiiiiiiiii	16-bit unsigned immediate data
u5	AAAAA0	5-bit unsigned SP address offset
s16	AAAAAAAAAAAAAAAA	16-bit signed SP address offset



JRA

Branch (AGU)

BRA

Operation

PC + displacement → PC

Assembler Syntax

BRA <label
BRA >label

Description

BRA >label

BRA <label

Causes program execution to continue at location PC + displacement. The displacement, calculated by the assembler and linker, is a two's complement integer that represents the relative distance from the current PC to the destination label. The assembler determines if the PC relative displacement is a short branch (<label [$-2^{10} \leq \text{displacement} < 2^{10}$, W]) or a long branch (>label [$-2^{20} \leq \text{displacement} < -2^{10}$, W and $2^{10} \leq \text{displacement} < 2^{20}$, W]).

Status and Conditions that Affect Instruction

None.

Status and Conditions Changed by Instruction

None.



xample

```

bra _label2 ; disassembled: bra >+*$8
nop
nop
_label2

```

Register/Memory Address	Before	After
_label (displacement)	\$0000 000A	
PC	\$0000 0002	\$0000 000A

Instruction Formats and Opcodes

Instruction	Words	Cycles	Type	Opcode												
BRA <label	1	4	4	<table border="1"> <tr> <td>15</td> <td>8</td> <td>7</td> <td>0</td> </tr> <tr> <td>1 0 0 0 1 A A A</td> <td>A A A A A A A</td> <td>1</td> <td></td> </tr> </table>	15	8	7	0	1 0 0 0 1 A A A	A A A A A A A	1					
15	8	7	0													
1 0 0 0 1 A A A	A A A A A A A	1														
BRA >label	2	4	4	<table border="1"> <tr> <td>15</td> <td>8</td> <td>7</td> <td>0</td> </tr> <tr> <td>0 0 1 0 a 0 0 1</td> <td>A A A 1 1 a a a</td> <td></td> <td></td> </tr> <tr> <td>1 0 0 A A A A A</td> <td>A A A A A A A</td> <td>a</td> <td></td> </tr> </table>	15	8	7	0	0 0 1 0 a 0 0 1	A A A 1 1 a a a			1 0 0 A A A A A	A A A A A A A	a	
15	8	7	0													
0 0 1 0 a 0 0 1	A A A 1 1 a a a															
1 0 0 A A A A A	A A A A A A A	a														

Instruction Fields

displacement (<label)	AAAAAAAAA0	10-bit signed PC relative displacement
displacement (>label)	aaaaAAAAAAAAAAAAAAAAA0	20-bit signed PC relative displacement



Operation

PC + displacement → PC

Assembler Syntax

```
BRAD <label
BRAD >label
```

Description

BRAD <label

BRAD >label

Causes program execution to continue at location PC + displacement after executing the execution set immediately following the execution set containing the BRAD instruction (called the delay slot). The displacement, calculated by the assembler and linker, is a two's complement integer that represents the relative distance from the current PC to the destination label. The assembler determines if the PC relative displacement is a short branch (<label [$-2^{10} \leq \text{displacement} < 2^{10}$, W]) or a long branch (>label [$-2^{20} \leq \text{displacement} < -2^{10}$, W and $2^{10} \leq \text{displacement} < 2^{20}$, W]).

Status and Conditions that Affect Instruction

None.

Status and Conditions Changed by Instruction

None.

Example

Source Code	Comments
move.l #\$1234,d0.l ;loads d0	
brad lbl3 ; disassembled code - brad >+a; p:lbl3	
add d0,d1,d7 ; executes, d7 = \$1234, pc then branches to \$e, address of lbl3	
nop	
nop	
lbl3 add d0,d7,d7 ; executes, d7 = \$2468	

Register/Memory Address	Before	After
lbl3 (displacement)	\$0000 000A	
PC	\$0000 0004	\$0000 000E

Instruction Formats and Opcodes

Instruction	Words	Cycles	Type	Opcode												
BRAD <label	1	4 ¹	4	<table border="1"> <tr> <td style="text-align: center;">15</td> <td style="text-align: center;">8</td> <td style="text-align: center;">7</td> <td style="text-align: center;">0</td> </tr> <tr> <td>1 0 0 0 1 A A A</td> <td>A A A A A A A</td> <td>A A A A A A A</td> <td>0</td> </tr> </table>	15	8	7	0	1 0 0 0 1 A A A	A A A A A A A	A A A A A A A	0				
15	8	7	0													
1 0 0 0 1 A A A	A A A A A A A	A A A A A A A	0													
BRAD >label	2	4 ¹	4	<table border="1"> <tr> <td style="text-align: center;">15</td> <td style="text-align: center;">8</td> <td style="text-align: center;">7</td> <td style="text-align: center;">0</td> </tr> <tr> <td>0 0 1 0 a 0 0 0</td> <td>A A A 1 1 a a a</td> <td>A A A A A A A</td> <td>a</td> </tr> <tr> <td>1 0 0 A A A A A A</td> <td>A A A A A A A</td> <td>A A A A A A A</td> <td>a</td> </tr> </table>	15	8	7	0	0 0 1 0 a 0 0 0	A A A 1 1 a a a	A A A A A A A	a	1 0 0 A A A A A A	A A A A A A A	A A A A A A A	a
15	8	7	0													
0 0 1 0 a 0 0 0	A A A 1 1 a a a	A A A A A A A	a													
1 0 0 A A A A A A	A A A A A A A	A A A A A A A	a													

Note 1: The branch uses 4 cycles minus the execution time used by execution set in the delay slot. The cycle count for this instruction cannot be less than 1 cycle.

Instruction Fields

displacement (<label)	AAAAAAAAAA0	10-bit signed PC relative displacement
displacement (>label)	aaaaaAAAAAAAAAAAAAAAAA0	20-bit signed PC relative displacement



BREAK

Terminate the Loop and Branch to an Address (AGU)

BREAK

Operation

PC + displacement → PC
0 → LFn

Assembler Syntax

BREAK label

Description

BREAK label

Exits the active loop n unconditionally before the active loop counter (LCn) equals one, and clears the active loop flag. The program execution continues at “label.” The displacement, calculated by the assembler and linker, is a two’s complement integer that represents the relative distance from the current PC to the destination label. Some programming rules apply to the use of this instruction. If no loops are enabled, this instruction is undefined.

Status and Conditions that Affect Instruction

Register Address	Bit Name	Description
SR[30:27]	LF[3:0]	Determines which loop is active.

Status and Conditions Changed by Instruction

Register Address	Bit Name	Description
SR[30:27]	LF[3:0]	Clear active loop flag.

Example

```
break _label
```

Register/Memory Address	Before	After
_label (displacement)	\$0000 000C	
PC	\$0000 0014	\$0000 0020
SR	\$40E0 0000	\$00E0 0000

Note: The assembler has calculated the displacement \$C to increment the program counter from its value at the BREAK (\$0000 0014) to its value at _label (\$0000 0020). LF3 is cleared by the break.



Instruction Formats and Opcodes

Instruction	Words	Cycles	Type	Opcode																																		
BREAK label	2	4	4	<div style="display: flex; justify-content: space-between; width: 100%;"> 15 8 7 0 </div> <table border="1" style="width: 100%; border-collapse: collapse;"> <tr> <td style="width: 8px;">0</td><td style="width: 8px;">0</td><td style="width: 8px;">1</td><td style="width: 8px;">0</td><td style="width: 8px;">0</td><td style="width: 8px;">0</td><td style="width: 8px;">0</td><td style="width: 8px;">0</td><td style="width: 8px;">0</td> <td style="width: 8px;">A</td><td style="width: 8px;">A</td><td style="width: 8px;">A</td><td style="width: 8px;">0</td><td style="width: 8px;">0</td><td style="width: 8px;">0</td><td style="width: 8px;">1</td><td style="width: 8px;">1</td> </tr> <tr> <td style="width: 8px;">1</td><td style="width: 8px;">0</td><td style="width: 8px;">0</td><td style="width: 8px;">A</td><td style="width: 8px;">A</td><td style="width: 8px;">A</td><td style="width: 8px;">A</td><td style="width: 8px;">A</td><td style="width: 8px;">A</td> <td style="width: 8px;">A</td><td style="width: 8px;">A</td><td style="width: 8px;">A</td><td style="width: 8px;">A</td><td style="width: 8px;">A</td><td style="width: 8px;">A</td><td style="width: 8px;">A</td><td style="width: 8px;">a</td> </tr> </table>	0	0	1	0	0	0	0	0	0	A	A	A	0	0	0	1	1	1	0	0	A	A	A	A	A	A	A	A	A	A	A	A	A	a
0	0	1	0	0	0	0	0	0	A	A	A	0	0	0	1	1																						
1	0	0	A	A	A	A	A	A	A	A	A	A	A	A	A	a																						

Instruction Fields

displacement aAAAAAAAAAAAAAAAAA0 16-bit signed PC relative displacement.
The encoding is the displacement with bit 0 stripped and replaced by the sign bit.



Operation

(Next PC) \rightarrow (SP); SR \rightarrow (SP + 4); SP + 8 \rightarrow SP;
PC + displacement \rightarrow PC; (Next PC) \rightarrow RAS

Assembler Syntax

BSR <label
BSR >label

Description

BSR <label

BSR >label

Pushes the next PC and SR onto the stack and causes program execution to continue at location PC + displacement. The displacement, calculated by the assembler and linker, is a two's complement integer that represents the relative distance from the current PC to the destination label. The assembler determines if the PC relative displacement is a short branch (<label [$-2^8 \leq \text{displacement} < 2^8$, W]) or a long branch (>label [$-2^{20} \leq \text{displacement} < -2^8$, W and $2^8 \leq \text{displacement} < 2^{20}$, W]). In addition to being pushed onto the stack, the next PC is stored in the return address from subroutine register (RAS) and RAS becomes valid.

Status and Conditions that Affect Instruction

Register Address	Bit Name	Description
SR[18]	EXP	Determines the stack pointer used in instructions that have a stack pointer as an operand.

Status and Conditions Changed by Instruction

None.

Example

```
bsr _label
```

Register/Memory Address	Before	After
SR	\$00E0 0000	
_label (displacement)	\$0000 0014	
PC	\$0000 0002	\$0000 0016
NSP	\$30	\$38
(\$30)		\$0000 0006
(\$34)		\$00E0 0000

BSRD Branch to Subroutine Using a Delay Slot (AGU) BSRD

Operation

(next* PC) → (SP); SR → (SP + 4); SP + 8 → SP;
PC + displacement → PC, (next* PC) → RAS

Assembler Syntax

BSRD <label
BSRD >label

Description

BSRD <label

BSRD >label

Executes the execution set in the delay slot, then pushes the next* PC (the PC of the execution set after the delay slot) and SR onto the stack, and causes program execution to continue at location PC + displacement. The displacement, calculated by the assembler and linker, is a two's complement integer that represents the relative distance from the current PC to the destination label. The assembler and linker determines if the PC relative displacement is a short branch (<label [$-2^8 \leq \text{displacement} < 2^8, W$]) or a long branch (>label [$-2^{20} \leq \text{displacement} < -2^8, W$ and $2^8 \leq \text{displacement} < 2^{20}, W$]). In addition to being pushed onto the stack, the next* PC is stored in the RAS register, and RAS becomes valid.

Status and Conditions that Affect Instruction

Register Address	Bit Name	Description
SR[18]	EXP	Determines the stack pointer used in instructions that have a stack pointer as an operand.

Status and Conditions Changed by Instruction

None.

Example

Source Code	Comments
move.l #\$30,r0	; loads r0 to later initialize sp
move.l #\$40,r1	; loads r1 to later initialize osp
tfra r0,sp	; initializes sp, sp is esp in this example
tfra r1,osp	; initializes osp, osp is nsp
nop	
bsrd lbl3	; branch to lbl3
move.w #\$1234,r0	; execute before the branch
nop	
lbl3 add d0,d1,d2	

Instruction Formats and Opcodes

Instruction	Words	Cycles ¹	Type	Opcode												
BSRD <label	1	4/5	4	<table border="1"> <tr> <td style="text-align: right;">15</td> <td style="text-align: center;">8</td> <td style="text-align: center;">7</td> <td style="text-align: right;">0</td> </tr> <tr> <td>1 0 0 0 0 0 1 A</td> <td>A</td> <td>A</td> <td>A A A A A A A 0</td> </tr> </table>	15	8	7	0	1 0 0 0 0 0 1 A	A	A	A A A A A A A 0				
15	8	7	0													
1 0 0 0 0 0 1 A	A	A	A A A A A A A 0													
BSRD >label	2	4/5	4	<table border="1"> <tr> <td style="text-align: right;">15</td> <td style="text-align: center;">8</td> <td style="text-align: center;">7</td> <td style="text-align: right;">0</td> </tr> <tr> <td>0 0 1 0 a 0 1 0</td> <td>A</td> <td>A</td> <td>A 1 1 a a a</td> </tr> <tr> <td>1 0 0 A A A A A</td> <td>A</td> <td>A</td> <td>A A A A A A a</td> </tr> </table>	15	8	7	0	0 0 1 0 a 0 1 0	A	A	A 1 1 a a a	1 0 0 A A A A A	A	A	A A A A A A a
15	8	7	0													
0 0 1 0 a 0 1 0	A	A	A 1 1 a a a													
1 0 0 A A A A A	A	A	A A A A A A a													

Note 1: The branch uses 4 cycles minus the execution time used by the execution set in the delay slot. The cycle count for this instruction cannot be less than 2 cycles. The branch uses 5 cycles, minus the execution time used by the execution set in the delay slot, if the total of the largest cycle time of the instructions grouped with the BSRD and the execution time of the delay slot set is ≥ 4 . One cycle is used by the core to push the return address onto the stack.

Instruction Fields

displacement (<label)	AAAAAAAA0	8-bit signed PC relative displacement
displacement (>label)	aaaaAAAAAAAAAAAAA A0	20-bit signed PC relative displacement

Operation

If $T=1$, then $PC + \text{displacement} \rightarrow PC$

Assembler Syntax

BT <label
BT >label

Description

BT <label

BT >label

Branches to label if the true bit is set. If the T bit is set, the program continues executing at location $PC + \text{displacement}$. If the T bit is cleared, the PC is updated to point to the next execution set, and the program continues executing sequentially. The displacement, calculated by the assembler and linker, is a two's complement integer that represents the relative distance from the current PC to the destination label. The assembler determines if the PC relative displacement is a short branch (<label [$-2^8 \leq \text{displacement} < 2^8, W$]) or a long branch (>label [$-2^{20} \leq \text{displacement} < -2^8, W$ and $2^8 \leq \text{displacement} < 2^{20}, W$]).

Status and Conditions that Affect Instruction

Register Address	Bit Name	Description
SR[1]	T	True bit

Status and Conditions Changed by Instruction

None.

Example

BT lbl

Instruction	Result
cmpeq.w #35,d1	Equal, so T bit in SR set.
bt lbl move.w #29,d1	Branch taken, move.w executed, d1=\$29.
inc d1	Skipped over.
move.w #47,d2	Skipped over.
- - - -	Skipped over.
- - - -	Skipped over.
- - - -	Skipped over.
lbl move.w #16,d4	Execution continues here at lbl, d4=\$16.

Register/Memory Address	Before BT	After
SR	\$00E4 0002	
d1	\$0035	\$0029

Register/Memory Address	Before BT	After
d2	\$0000	\$0000
pc	\$0006	\$0016

Instruction Formats and Opcodes

Instruction	Words	Cycles ¹	Type	Opcode																																				
BT <label	1	1/4	4	<table border="1"> <tr> <td>15</td> <td>8</td> <td>7</td> <td>0</td> </tr> <tr> <td>1</td><td>0</td><td>0</td><td>0</td> <td>0</td><td>0</td><td>0</td><td>A</td> <td>A</td><td>A</td><td>A</td><td>A</td> <td>A</td><td>A</td><td>A</td><td>1</td> </tr> </table>	15	8	7	0	1	0	0	0	0	0	0	A	A	A	A	A	A	A	A	1																
15	8	7	0																																					
1	0	0	0	0	0	0	A	A	A	A	A	A	A	A	1																									
BT >label	2	1/4	4	<table border="1"> <tr> <td>15</td> <td>8</td> <td>7</td> <td>0</td> </tr> <tr> <td>0</td><td>0</td><td>1</td><td>0</td> <td>a</td><td>1</td><td>0</td><td>1</td> <td>A</td><td>A</td><td>A</td><td>1</td> <td>1</td><td>a</td><td>a</td><td>a</td> </tr> <tr> <td>1</td><td>0</td><td>0</td><td>A</td> <td>A</td><td>A</td><td>A</td><td>A</td> <td>A</td><td>A</td><td>A</td><td>A</td> <td>A</td><td>A</td><td>A</td><td>a</td> </tr> </table>	15	8	7	0	0	0	1	0	a	1	0	1	A	A	A	1	1	a	a	a	1	0	0	A	A	A	A	A	A	A	A	A	A	A	A	a
15	8	7	0																																					
0	0	1	0	a	1	0	1	A	A	A	1	1	a	a	a																									
1	0	0	A	A	A	A	A	A	A	A	A	A	A	A	a																									

Note 1: If the branch is not taken, it uses 1 cycle. If the branch is taken, it uses 4 cycles.

Instruction Fields

displacement (<label)	AAAAAAAAA0	8-bit signed PC relative displacement
displacement (>label)	aaaaaAAAAAAAAAAAAAAAAA0	20-bit signed PC relative displacement



Operation

If $T=1$, then $PC + displacement \rightarrow PC$

Assembler Syntax

```
BTD <label
BTD >label
```

Description

BTD <label

BTD >label

Branches to label if the true bit is set. If the T bit is set, the program continues executing at location $PC + displacement$. If the T bit is cleared, the PC is updated to point to the next execution set, and the program continues executing sequentially. The displacement, calculated by the assembler and linker, is a two's complement integer that represents the relative distance from the current PC to the destination label. The assembler determines if the PC relative displacement is a short branch ($\text{<label} [-2^8 \leq \text{displacement} < 2^8, W]$) or a long branch ($\text{>label} [-2^{20} \leq \text{displacement} < -2^8, W \text{ and } 2^8 \leq \text{displacement} < 2^{20}, W]$). The execution set in the delay slot immediately following the BTD instruction is executed unconditionally after the execution set containing the BTD instruction.

Status and Conditions that Affect Instruction

Register Address	Bit Name	Description
SR[1]	T	True bit

Status and Conditions Changed by Instruction

None.

Example

```
BTD lbl
```

Instruction	Result
<code>cmpeq.w #\$35,d1</code>	Equal, so T bit in SR set.
<code>btd lbl move.w #\$29,d1</code>	Branch taken, <code>move.w</code> executed, $d1 = \$29$.
<code>inc d1</code>	Increment executed in the delay slot, $d1 = \$2A$.
<code>move.w #\$47,d2</code>	Skipped over.
----	Skipped over.
----	Skipped over.
----	Skipped over.
<code>lbl move.w #\$1A,d4</code>	Execution continues here at <code>lbl</code> .

Register/Memory Address	Before BTD	After
SR	\$00E0 0002	

Register/Memory Address	Before BTD	After
d1	\$0035	\$002A
d2	\$0000	\$0000
d4	\$0000	\$001A
pc	\$0006	\$0016

Instruction Formats and Opcodes

Instruction	Words	Cycles ¹	Type	Opcode												
BTD <label	1	1/4	4	<table border="1"> <tr> <td style="text-align: right;">15</td> <td style="text-align: right;">8</td> <td style="text-align: right;">7</td> <td style="text-align: right;">0</td> </tr> <tr> <td>1 0 0 0 0 0 0 A</td> <td>A A A A A A A</td> <td>A</td> <td>0</td> </tr> </table>	15	8	7	0	1 0 0 0 0 0 0 A	A A A A A A A	A	0				
15	8	7	0													
1 0 0 0 0 0 0 A	A A A A A A A	A	0													
BTD >label	2	1/4	4	<table border="1"> <tr> <td style="text-align: right;">15</td> <td style="text-align: right;">8</td> <td style="text-align: right;">7</td> <td style="text-align: right;">0</td> </tr> <tr> <td>0 0 1 0 a 1 0 0</td> <td>A A A 1 1 a a a</td> <td></td> <td></td> </tr> <tr> <td>1 0 0 A A A A A</td> <td>A A A A A A A</td> <td>A</td> <td>a</td> </tr> </table>	15	8	7	0	0 0 1 0 a 1 0 0	A A A 1 1 a a a			1 0 0 A A A A A	A A A A A A A	A	a
15	8	7	0													
0 0 1 0 a 1 0 0	A A A 1 1 a a a															
1 0 0 A A A A A	A A A A A A A	A	a													

Note 1: If the branch is not taken, it uses 1 cycle. If the branch is taken, it uses 4 cycles minus the time used by the execution set in the delay slot. The cycle count for this instruction cannot be less than 1 cycle.

Instruction Fields

displacement (<label)	AAAAAAAA0	8-bit signed PC relative displacement
displacement (>label)	aaaaaAAAAAAAAAAAAAAAAA0	20-bit signed PC relative displacement

Operation

If $Da[39] == 0$,
 then $9 - (\text{number of consecutive leading zeros in } Da[39:0]) \rightarrow Dn$
 else $9 - (\text{number of consecutive leading ones in } Da[39:0]) \rightarrow Dn$

Assembler Syntax

CLB Da, Dn

Description

CLB Da,Dn

Counts the leading 0s or 1s according to bit 39 of source Da. It scans bits [39:0] of Da starting from bit 39. The operation loads nine minus the number of consecutive leading 0s or 1s into destination Dn. The result is sign-extended. The range of the result is +8 to -31. This instruction can be used in conjunction with the instruction ASRR for performing fast normalization of the operand. If Da equals zero, then Dn is set to zero.

Status and Conditions that Affect Instruction

None.

Status and Conditions Changed by Instruction

Register Address	Bit Name	Description
Ln	L	Clear the Ln bit in the destination data register.

Example

```
c1b d3,d7
```

Register/Memory Address	Before	After
D3	\$00000 F7434	
L7:D7		\$0:\$FF FFFF FFF5

The number of consecutive zeros is 20, $9 - 20 = -11$ (\$FFF5)

Instruction Formats and Opcodes

Instruction	Words	Cycles	Type	Opcode																
CLB Da, Dn	1	1	2	<div style="display: flex; justify-content: space-between; align-items: center;"> 15 8 7 0 </div> <table border="1" style="width: 100%; text-align: center; border-collapse: collapse;"> <tr> <td style="width: 25%;">1</td> <td style="width: 25%;">1</td> <td style="width: 25%;">0</td> <td style="width: 25%;">1</td> <td style="width: 25%;">0</td> <td style="width: 25%;">0</td> <td style="width: 25%;">F</td> <td style="width: 25%;">F</td> <td style="width: 25%;">F</td> <td style="width: 25%;">0</td> <td style="width: 25%;">0</td> <td style="width: 25%;">1</td> <td style="width: 25%;">0</td> <td style="width: 25%;">J</td> <td style="width: 25%;">J</td> <td style="width: 25%;">J</td> </tr> </table>	1	1	0	1	0	0	F	F	F	0	0	1	0	J	J	J
1	1	0	1	0	0	F	F	F	0	0	1	0	J	J	J					

Instruction Fields

Da **JJJ** **Single Source Data Register**

000	D0	010	D2	100	D4	110	D6
001	D1	011	D3	101	D5	111	D7

Note: This instruction can specify D8-D15 as operands by using a prefix.

Dn **FFF** **Single Source/Destination Data Register**

000	D0	010	D2	100	D4	110	D6
001	D1	011	D3	101	D5	111	D7

Note: This instruction can specify D8-D15 as operands by using a prefix.



Operation

$0 \rightarrow D_n$

Assembler Syntax

CLR D_n

Description

CLR D_n

Clears a data register (D_n).

Note: CLR D_n is assembler mapped to SUB D_a, D_a, D_n where D_n is the register being cleared and D_a is an arbitrary register assigned by the assembler for programming rule G.G.5. Any ($D_a - D_a$) results in zero being stored in D_n . D_a assignment uses the low data registers ($D_0 - D_7$) where possible to avoid using a prefix.

Status and Conditions that Affect Instruction

None.

Status and Conditions Changed by Instruction

Register Address	Bit Name	Description
L_n	L	Clears the L_n bit in the destination register.
SR[0]	C	Clears the carry bit.

Example

```
clr d1
```

Register/Memory Address	Before	After
SR	\$00E0 0001	\$00E0 0000
L1:D1	\$0:\$00 0000 0040	\$0:\$00 0000 0000

Instruction Formats and Opcodes

Instruction	Words	Cycles	Type	Opcode
CLR D_n (D_a even)	1	1	1	<div style="display: flex; justify-content: space-between; align-items: center;"> 15 8 7 0 </div> <div style="display: flex; justify-content: space-between; align-items: center;"> 0 * 1 0 1 1 F F F 0 0 J J J J J </div>
CLR D_n (D_a odd)	1	1	1	<div style="display: flex; justify-content: space-between; align-items: center;"> 15 8 7 0 </div> <div style="display: flex; justify-content: space-between; align-items: center;"> 0 * 1 0 0 0 F F F 1 1 0 0 1 j j </div>

Note: ** indicates serial grouping encoding.

struction Fields

Dn **FFF** **Destination Data Register**

000	D0	010	D2	100	D4	110	D6
001	D1	011	D3	101	D5	111	D7

Note: This instruction can specify D8-D15 as operands by using a prefix.

Da **JJJJ** **Source Data Register**

10000	D0	11100	D2	10100	D4	11110	D6

Note: This instruction can specify D8-D15 as operands by using a prefix.

Da **jj** **Source Data Register**

00	D1	01	D3	10	D5	11	D7

Note: If registers D8–D15 are accessed instead of D0–D7, a prefix is used.



Operation

If $D_a == D_n$, then $1 \rightarrow T$, else $0 \rightarrow T$

Assembler Syntax

CMPEQ D_a, D_n

Description

CMPEQ D_a, D_n

Compares the 40-bit contents of two data registers (D_a and D_n), setting the T bit if they are equal, and clearing the T bit if they are not.

Status and Conditions that Affect Instruction

None.

Status and Conditions Changed by Instruction

Register Address	Bit Name	Description
SR[1]	T	Sets T bit if equal, otherwise cleared.

Example

```
cmpeq d2, d3
```

Register/Memory Address	Before	After
D2	\$00 0000 0005	
D3	\$00 0000 0005	
SR	\$00E4 0000	\$00E4 0002

Instruction Formats and Opcodes

Instruction	Words	Cycles	Type	Opcode																				
CMPEQ D_a, D_n	1	1	1	<table border="1" style="display: inline-table; vertical-align: middle;"> <tr> <td style="text-align: center;">15</td> <td style="text-align: center;">8</td> <td style="text-align: center;">7</td> <td style="text-align: center;">0</td> </tr> <tr> <td style="text-align: center;">0</td> <td style="text-align: center;">*</td> <td style="text-align: center;">1</td> <td style="text-align: center;">1</td> </tr> <tr> <td style="text-align: center;">0</td> <td style="text-align: center;">0</td> <td style="text-align: center;">F</td> <td style="text-align: center;">F</td> </tr> <tr> <td style="text-align: center;">F</td> <td style="text-align: center;">1</td> <td style="text-align: center;">1</td> <td style="text-align: center;">0</td> </tr> <tr> <td style="text-align: center;">0</td> <td style="text-align: center;">J</td> <td style="text-align: center;">J</td> <td style="text-align: center;">J</td> </tr> </table>	15	8	7	0	0	*	1	1	0	0	F	F	F	1	1	0	0	J	J	J
15	8	7	0																					
0	*	1	1																					
0	0	F	F																					
F	1	1	0																					
0	J	J	J																					

Note: ** indicates serial grouping encoding.

Instruction Fields

Da	Single Source Data Register						
	JJJ		JJJ		JJJ		
000	D0	010	D2	100	D4	110	D6
001	D1	011	D3	101	D5	111	D7

Note: This instruction can specify D8-D15 as operands by using a prefix.



CMPEQ.W

Compare for Equal (DALU)

CMPEQ.W

Operation

If $\#u5 == Dn$, then $1 \rightarrow T$, else $0 \rightarrow T$

If $\#s16 == Dn$, then $1 \rightarrow T$, else $0 \rightarrow T$

Assembler Syntax

`CMPEQ.W #u5,Dn` $\{0 \leq u5 < 32\}$

`CMPEQ.W #s16,Dn` $\{-2^{15} \leq s16 < 2^{15}\}$

Description

CMPEQ.W #u5,Dn

Compares an immediate unsigned 5-bit value (range 0–31) with a data register (Dn) for equality. The immediate value is right-aligned and zero-extended.

CMPEQ.W #s16,Dn

Compares an immediate signed 16-bit value that has been right-aligned and sign-extended to 40 bits with a data register (Dn) for equality.

Status and Conditions that Affect Instruction

None.

Status and Conditions Changed by Instruction

Register Address	Bit Name	Description
SR[1]	T	Sets T bit if equal, otherwise cleared.

Example

```
cmpeq.w #5,d3
```

Register/Memory Address	Before	After
immediate	\$0000 0005	
D3	\$00 0000 0005	
SR	\$00E4 0000	\$00E4 0002



Operation

If $rx == Rx$, then $1 \rightarrow T$, else $0 \rightarrow T$

Assembler Syntax

CMPEQA rx, Rx

Description

CMPEQA rx, Rx

Compares two AGU registers (rx and Rx) for equality. Note that a register cannot be compared to itself using this instruction.

Status and Conditions that Affect Instruction

Register Address	Bit Name	Description
SR[18]	EXP	Determines which stack pointer is used when the stack pointer is an operand. Otherwise, the instruction is not affected by SR.

Status and Conditions Changed by Instruction

Register Address	Bit Name	Description
SR[1]	T	Sets T bit if equal, otherwise cleared.

Example

```
cmpeqa r1,r2
```

Register/Memory Address	Before	After
R1	\$0000 0005	
R2	\$0000 0005	
SR	\$00E4 0000	\$00E4 0002

Instruction Formats and Opcodes

Instruction	Words	Cycles	Type	Opcode															
CMPEQA rx,Rx	1	1	2	<div style="display: flex; justify-content: space-between; align-items: center;"> 15 8 7 0 </div> <table border="1" style="width: 100%; text-align: center; border-collapse: collapse;"> <tr> <td style="width: 25%;">1</td> <td style="width: 25%;">1</td> <td style="width: 25%;">0</td> <td style="width: 25%;">R</td> <td style="width: 25%;">R</td> <td style="width: 25%;">R</td> <td style="width: 25%;">R</td> <td style="width: 25%;">1</td> <td style="width: 25%;">0</td> <td style="width: 25%;">1</td> <td style="width: 25%;">0</td> <td style="width: 25%;">r</td> <td style="width: 25%;">r</td> <td style="width: 25%;">r</td> <td style="width: 25%;">r</td> </tr> </table>	1	1	0	R	R	R	R	1	0	1	0	r	r	r	r
1	1	0	R	R	R	R	1	0	1	0	r	r	r	r					

Instruction Fields

rx	rrrr	AGU Source Register					
0000	N0	0100	—	1000	R0	1100	R4
0001	N1	0101	—	1001	R1	1101	R5
0010	N2	0110	PC	1010	R2	1110	R6
0011	N3	0111	SP	1011	R3	1111	R7

Note: This instruction can specify R8-R15 as operands by using a high register prefix.

Rx	RRRR	AGU Source/Destination Register					
0000	N0	0100	—	1000	R0	1100	R4
0001	N1	0101	—	1001	R1	1101	R5
0010	N2	0110	—	1010	R2	1110	R6
0011	N3	0111	SP	1011	R3	1111	R7

Note: This instruction can specify R8-R15 as operands by using a high register prefix.



CMPGT

Compare for Greater Than (DALU)

CMPGT

Operation

$D_n > D_a \rightarrow T$

Assembler Syntax

CMPGT D_a, D_n

Description

CMPGT D_a, D_n

Compares two data registers (D_a and D_n). The T bit is set if the signed value in the second data register (D_n) is greater than the signed value in the first (D_a); T is cleared otherwise.

Status and Conditions that Affect Instruction

None.

Status and Conditions Changed by Instruction

Register Address	Bit Name	Description
SR[1]	T	Sets T bit if $D_n > D_a$, otherwise cleared.

Example

```
cmpgt d2,d3
```

Register/Memory Address	Before	After
D2	\$0000 35FA	
D3	\$0000 35FB	
SR	\$00E4 0000	\$00E4 0002

Instruction Formats and Opcodes

Instruction	Words	Cycles	Type	Opcode
CMPGT D_a, D_n	1	1	1	<div style="display: flex; justify-content: space-between; align-items: center;"> 15 8 7 0 </div> <div style="border: 1px solid black; padding: 2px; display: flex; justify-content: space-between; align-items: center;"> 0 * 1 1 0 0 F F F 1 1 1 0 J J J </div>

Note: ** indicates serial grouping encoding.

Instruction Fields

Da	Single Source Data Register						
	JJJ		JJJ		JJJ		
000	D0	010	D2	100	D4	110	D6
001	D1	011	D3	101	D5	111	D7

Note: This instruction can specify D8-D15 as operands by using a prefix.



CMPGT.W

Compare for Greater Than (DALU)

CMPGT.W

Operation

$Dn > \#u5 \rightarrow T$

$Dn > \#s16 \rightarrow T$

Assembler Syntax

`CMPGT.W #u5,Dn {0 ≤ u5 < 32}`

`CMPGT.W #s16,Dn {-215 ≤ s16 < 215}`

Description

These instructions set the T bit if the content of a signed data register (Dn) is greater than the immediate value, or clear the T bit if the content of the data register is not greater than the immediate value.

CMPGT.W #u5,Dn

Compares if a data register is greater than an immediate unsigned 5-bit value that has been right-aligned and zero-extended to 40 bits.

CMPGT.W #s16,Dn

Compares if a data register is greater than an immediate signed 16-bit value that has been right-aligned and sign-extended to 40 bits.

Status and Conditions that Affect Instruction

None.

Status and Conditions Changed by Instruction

Register Address	Bit Name	Description
SR[1]	T	Sets T bit if $Dn >$ an immediate, otherwise cleared.

Example

```
cmpgt.w #8002,d2
```

Register/Memory Address	Before	After
immediate	\$FF FFFF 8002	
D2	\$FF FFFF 8004	
SR	\$00E4 0000	\$00E4 0002



CMPGTA Compare for Greater Than (AGU) CMPGTA

Operation

$Rx > rx \rightarrow T$

Assembler Syntax

CMPGTA rx,Rx

Description

CMPGTA rx,Rx

Compares two signed AGU registers (rx and Rx) and sets the T bit if the second AGU register is greater than the first, or clears the T bit if the second AGU register is not greater than the first. Note that a register cannot be compared to itself using this instruction.

Status and Conditions that Affect Instruction

Register Address	Bit Name	Description
SR[18]	EXP	Determines which stack pointer is used when the stack pointer is an operand. Otherwise, the instruction is not affected by SR.

Status and Conditions Changed by Instruction

Register Address	Bit Name	Description
SR[1]	T	Sets the T bit if $Rx > rx$, otherwise cleared.

Example

```
cmpgta r2,r3
```

Register/Memory Address	Before	After
R2	\$0000 35FA	
R3	\$0000 34EA	
SR	\$00E4 0002	\$00E4 0000

Instruction Formats and Opcodes

Instruction	Words	Cycles	Type	Opcode
CMPGTA rx,Rx	1	1	2	<div style="display: flex; justify-content: space-between; align-items: center;"> 15 8 7 0 </div> <div style="border: 1px solid black; padding: 2px; display: flex; justify-content: space-between; align-items: center;"> 1 1 1 0 R R R R 1 0 0 1 r r r r </div>

Instruction Fields

rx	rrrr	AGU Source Register					
0000	N0	0100	—	1000	R0	1100	R4
0001	N1	0101	—	1001	R1	1101	R5
0010	N2	0110	PC	1010	R2	1110	R6
0011	N3	0111	SP	1011	R3	1111	R7

Note: This instruction can specify R8-R15 as operands by using a high register prefix.

Rx	RRRR	AGU Source/Destination Register					
0000	N0	0100	—	1000	R0	1100	R4
0001	N1	0101	—	1001	R1	1101	R5
0010	N2	0110	—	1010	R2	1110	R6
0011	N3	0111	SP	1011	R3	1111	R7

Note: This instruction can specify R8-R15 as operands by using a high register prefix.



Operation

$D_n > D_a \rightarrow T$

Assembler Syntax

`CMPHI Da, Dn`

Description

CMPHI Da, Dn

Compares the unsigned value in bits 31:0 of two data registers (D_a and D_n) to determine which is greater. It sets the T bit if the unsigned value of $D_n[31:0]$ is greater than the unsigned value of $D_a[31:0]$. Otherwise, it clears the T bit.

Status and Conditions that Affect Instruction

None.

Status and Conditions Changed by Instruction

Register Address	Bit Name	Description
SR[1]	T	Sets the T bit if 32-bit unsigned $D_n > D_a$, otherwise cleared.

Example

```
cmphi d1, d0
```

Register/Memory Address	Before	After
D1	\$00 26A2 44F3	
D0	\$00 2781 21A2	
SR	\$00E4 0000	\$00E4 0002

Instruction Formats and Opcodes

Instruction	Words	Cycles	Type	Opcode
CMPHI Da, Dn	1	1	1	<div style="display: flex; justify-content: space-between; align-items: center;"> 15 8 7 0 </div> <div style="border: 1px solid black; padding: 2px; display: flex; justify-content: space-between; align-items: center;"> 0 * 1 1 0 0 F F F 1 1 1 1 J J J </div>

Note: ** indicates serial grouping encoding.

Instruction Fields

Da

JJJ

Single Source Data Register

000	D0	010	D2	100	D4	110	D6
001	D1	011	D3	101	D5	111	D7

Note: This instruction can specify D8-D15 as operands by using a prefix.

FFF**Single Source/Destination Data Register**

000	D0	010	D2	100	D4	110	D6
001	D1	011	D3	101	D5	111	D7

Note: This instruction can specify D8-D15 as operands by using a prefix.



Operation

$Rx > rx \rightarrow T$

Assembler Syntax

CMPHIA rx,Rx

Description

CMPHIA rx,Rx

Compares the unsigned value in two AGU registers (rx and Rx) to determine which is greater. It sets the T bit if the unsigned value of Rx is greater than the unsigned value of rx. It clears the T bit if the unsigned value of Rx is not greater than the unsigned value of rx. Note that a register cannot be compared to itself using this instruction.

Status and Conditions that Affect Instruction

Register Address	Bit Name	Description
SR[18]	EXP	Determines which stack pointer is used when the stack pointer is an operand. Otherwise, the instruction is not affected by SR.

Status and Conditions Changed By Instruction

Register Address	Bit Name	Description
SR[1]	T	Sets the T bit if unsigned $Rx > rx$, otherwise cleared.

Example

```
cmphia r0,r1
```

Register/Memory Address	Before	After
R0	\$FFFF 8002	
R1	\$FFFF FFFF	
SR	\$00E4 0000	\$00E4 0002



Instruction Formats and Opcodes

Instruction	Words	Cycles	Type	Opcode																
CMPHIA rx,Rx	1	1	2	<div style="display: flex; justify-content: space-between; align-items: center;"> 15 8 7 0 </div> <table border="1" style="width: 100%; text-align: center;"> <tr> <td>1</td><td>1</td><td>1</td><td>0</td><td>R</td><td>R</td><td>R</td><td>R</td> <td>1</td><td>0</td><td>0</td><td>0</td><td>r</td><td>r</td><td>r</td><td>r</td> </tr> </table>	1	1	1	0	R	R	R	R	1	0	0	0	r	r	r	r
1	1	1	0	R	R	R	R	1	0	0	0	r	r	r	r					

Instruction Fields

rx	rrrr	AGU Source Register					
0000	N0	0100	—	1000	R0	1100	R4
0001	N1	0101	—	1001	R1	1101	R5
0010	N2	0110	PC	1010	R2	1110	R6
0011	N3	0111	SP	1011	R3	1111	R7

Note: This instruction can specify R8-R15 as operands by using a high register prefix.

Rx	RRRR	AGU Source/Destination Register					
0000	N0	0100	—	1000	R0	1100	R4
0001	N1	0101	—	1001	R1	1101	R5
0010	N2	0110	—	1010	R2	1110	R6
0011	N3	0111	SP	1011	R3	1111	R7

Note: This instruction can specify R8-R15 as operands by using a high register prefix.

CONT**Continue to the Next Loop Iteration (AGU)****CONT****Operation****Assembler Syntax**

If $LC_n > 1$, then $SA_n \rightarrow PC$, $LC_{n-1} \rightarrow LC_n$ `CONT label`
 else $PC + \text{displacement} \rightarrow PC$
 $0 \rightarrow LFn$,
 $0 \rightarrow LC_n$

Description**CONT label**

Continues the active loop n from the start address of the loop (SA_n) if its loop counter (LC_n) is greater than one. Otherwise, it clears the active loop flag (LF_n) and branches to an address determined by a 16-bit signed displacement [$-2^{16} \leq \text{displacement} < 2^{16}, w$] added to the PC. In either case, the active loop counter is decremented by one. Some programming rules apply to the use of this instruction. If no loops are enabled, this instruction is undefined.

Status and Conditions that Affect Instruction

Register Address	Bit Name	Description
SR[30:27]	LF[3:0]	Read loop flags to determine active loop.

Status and Conditions Changed by Instruction

Register Address	Bit Name	Description
SR[30:27]	LF[3:0]	Clear active loop flag if the active loop counter is less than or equal to one.

Example

```
cont _label
```

Register/Memory Address	Before	After
_label (displacement)	\$C	
LC3	\$1	\$0
SR	\$40E4 0000	\$00E4 0000
PC	\$0000 0014	\$0000 0020

Loop count 3 (LC3) is 1, so loop count is decremented to 0, loop flag 3 (SR26) is cleared, and program continues at `_label`, address \$0000 0020.



Instruction Formats and Opcodes

Instruction	Words	Cycles ¹	Type	Opcode																																
CONT label	2	3/4	4	<div style="display: flex; justify-content: space-between; width: 100%;"> 15 8 7 0 </div> <table border="1" style="width: 100%; border-collapse: collapse;"> <tr> <td style="width: 8px;">0</td><td style="width: 8px;">0</td><td style="width: 8px;">1</td><td style="width: 8px;">0</td><td style="width: 8px;">0</td><td style="width: 8px;">0</td><td style="width: 8px;">1</td><td style="width: 8px;">1</td> <td style="width: 8px;">A</td><td style="width: 8px;">A</td><td style="width: 8px;">A</td><td style="width: 8px;">0</td><td style="width: 8px;">0</td><td style="width: 8px;">0</td><td style="width: 8px;">1</td><td style="width: 8px;">1</td> </tr> <tr> <td style="width: 8px;">1</td><td style="width: 8px;">0</td><td style="width: 8px;">0</td><td style="width: 8px;">A</td><td style="width: 8px;">A</td><td style="width: 8px;">A</td><td style="width: 8px;">A</td><td style="width: 8px;">A</td> <td style="width: 8px;">A</td><td style="width: 8px;">A</td><td style="width: 8px;">A</td><td style="width: 8px;">A</td><td style="width: 8px;">A</td><td style="width: 8px;">A</td><td style="width: 8px;">A</td><td style="width: 8px;">a</td> </tr> </table>	0	0	1	0	0	0	1	1	A	A	A	0	0	0	1	1	1	0	0	A	A	A	A	A	A	A	A	A	A	A	A	a
0	0	1	0	0	0	1	1	A	A	A	0	0	0	1	1																					
1	0	0	A	A	A	A	A	A	A	A	A	A	A	A	a																					

Note 1: If LC > 1, CONT uses 3 cycles. If LC ≤ 1, CONT uses 4 cycles.

Instruction Fields

displacement aAAAAAAAAAAAAAAAAA0 16-bit signed PC relative displacement. The encoding is the displacement with bit 0 stripped and replaced by the sign bit.



CONTD

Continue to Next Loop Iteration Using a Delay Slot (AGU)

CONTD

Operation

If $LC_n > 1$, then $SA_n \rightarrow PC$, $LC_{n-1} \rightarrow LC_n$
 else $PC + displacement \rightarrow PC$
 $0 \rightarrow LFn$,
 $0 \rightarrow LC_n$

Assembler Syntax

CONTD label

Description

CONTD label

Continues the active loop n from the start address of the loop (SA_n) if its loop counter (LC_n) is greater than one. Otherwise, it clears the active loop flag (LF_n), and branches to an address determined by a 16-bit signed displacement [$-2^{16} \leq displacement < 2^{16}, w$] added to the PC. In either case, the active loop counter is decremented by one, and the execution set immediately following the execution set containing the CONTD is executed. Some programming rules apply to the use of this instruction. If no loops are enabled, this instruction is undefined.

Status and Conditions that Affect Instruction

Register Address	Bit Name	Description
SR[30:27]	LF[3:0]	Read loop flags to determine active loop.

Status and Conditions Changed by Instruction

Register Address	Bit Name	Description
SR[30:27]	LF[3:0]	Clear active loop flag if the active loop counter is less than or equal to one.

Example

```
contd lbl3
```

Instruction	Result
dosetup strt0	; defines start address for loop 0
doen0 #10	; activates loop 0 with a count of 16
loopstart0	; assembler directive defining starting address SA
strt0 mac d0,d1,d2	; DALU instruction at start address
add d5,d6,d7	
contd lbl3	; PC returns to strt0 until LC = 1
inc d1	; executes in the delay slot each time, PC jumps to lbl3 when LC = 1
nop	
nop	



loopend0

lbl3 add d0,d1,d2

Instruction Formats and Opcodes

Instruction	Words	Cycles ¹	Type	Opcode																																
CONTD label	2	3/4	4	<div style="display: flex; justify-content: space-between; margin: 0;"> 15 8 7 0 </div> <table border="1" style="width: 100%; border-collapse: collapse; text-align: center;"> <tr> <td style="width: 8px;">0</td><td style="width: 8px;">0</td><td style="width: 8px;">1</td><td style="width: 8px;">0</td><td style="width: 8px;">0</td><td style="width: 8px;">0</td><td style="width: 8px;">1</td><td style="width: 8px;">0</td> <td style="width: 8px;">A</td><td style="width: 8px;">A</td><td style="width: 8px;">A</td><td style="width: 8px;">0</td><td style="width: 8px;">0</td><td style="width: 8px;">0</td><td style="width: 8px;">1</td><td style="width: 8px;">1</td> </tr> <tr> <td style="width: 8px;">1</td><td style="width: 8px;">0</td><td style="width: 8px;">0</td><td style="width: 8px;">A</td><td style="width: 8px;">A</td><td style="width: 8px;">A</td><td style="width: 8px;">A</td><td style="width: 8px;">A</td> <td style="width: 8px;">A</td><td style="width: 8px;">A</td><td style="width: 8px;">A</td><td style="width: 8px;">A</td><td style="width: 8px;">A</td><td style="width: 8px;">A</td><td style="width: 8px;">A</td><td style="width: 8px;">a</td> </tr> </table>	0	0	1	0	0	0	1	0	A	A	A	0	0	0	1	1	1	0	0	A	A	A	A	A	A	A	A	A	A	A	A	a
0	0	1	0	0	0	1	0	A	A	A	0	0	0	1	1																					
1	0	0	A	A	A	A	A	A	A	A	A	A	A	A	a																					

Note 1: If LC > 1, CONTD uses 3 cycles. If LC = 1, CONTD uses 4 cycles. In both cases, the cycles are decreased by the time used for the execution set in the delay slot. The cycle count for this instruction cannot be less than 1 cycle.

Instruction Fields

displacement aAAAAAAAAAAAAAAAAA0 16-bit signed PC relative displacement. The encoding is the displacement with bit 0 stripped and replaced by the sign bit.



DEBUG

Enter Debug Mode (AGU)

DEBUG

Operation

Assembler Syntax

DEBUG

Description

DEBUG

Causes the device to enter the debug state. It is an Enhanced On-chip Emulator (EOnCE) dedicated instruction that is used for debugging. This instruction cannot be grouped with another debug instruction.

Status and Conditions that Affect Instruction

None

Status and Conditions Changed by Instruction

None.

Example

debug

Instruction Formats and Opcodes

Instruction	Words	Cycles	Type	Opcode																							
DEBUG	1	2	4	<table border="1"> <tr> <td style="text-align: center;">15</td> <td></td> <td style="text-align: center;">8</td> <td style="text-align: center;">7</td> <td></td> <td style="text-align: center;">0</td> </tr> <tr> <td>1</td> <td>0</td> <td>0</td> <td>1</td> <td>1</td> <td>1</td> <td>1</td> <td>1</td> <td>0</td> <td>0</td> <td>1</td> <td>1</td> <td>1</td> <td>0</td> <td>0</td> <td>0</td> <td>0</td> </tr> </table>	15		8	7		0	1	0	0	1	1	1	1	1	0	0	1	1	1	0	0	0	0
15		8	7		0																						
1	0	0	1	1	1	1	1	0	0	1	1	1	0	0	0	0											



Operation

Assembler Syntax

DEBUGEV

Description

DEBUGEV

Generates a debug event. It is an EOnCE dedicated instruction. If the EOnCE has not been enabled since reset, issuing DEBUGEV has no effect. If the EOnCE is enabled, the effect of this instruction depends on the programming of EOnCE control registers. Receipt of an EOnCE event can cause the core to enter the debug mode, generate an exception, or enable the trace buffer. The delay from the DEBUGEV instruction to entering debug mode or generating an exception is not precise; it could be a few execution sets. Events can also be counted before an action takes place.

Status and Conditions that Affect Instruction

None.

Status and Conditions Changed by Instruction

None.

Instruction Formats and Opcodes

Instruction	Words	Cycles	Type	Opcode
DEBUGEV	1	2	4	<div style="display: flex; justify-content: space-between; align-items: center;"> 15 8 7 0 </div> <div style="border: 1px solid black; padding: 2px; display: flex; justify-content: space-between;"> 1 0 0 1 1 1 1 0 0 1 1 1 0 0 0 1 </div>



Operation

$Rx - 1 \rightarrow Rx$

Assembler Syntax

DECA Rx

Description

DECA Rx

Subtracts one from an AGU register (Rx). SP cannot be used as a destination of this instruction.

Note: The assembler maps this instruction to SUBA #u5,Rx; where #u5 = 1.

Status and Conditions that Affect Instruction

Register Address	Bit Name	Description
MCTL[31:0]	AM3–AM0	Address modification bits when updating R0–R7. Otherwise, the instruction is not affected by MCTL.

Status and Conditions Changed by Instruction

None.

Example

deca r0

Register/Memory Address	Before	After
MCTL	\$0000 0000	
R0	\$074F 312A	\$074F 3129

Instruction Formats and Opcodes

Instruction	Words	Cycles	Type	Opcode
DECA Rx	1	1	2	<div style="display: flex; justify-content: space-between; align-items: center;"> 15 8 7 0 </div> <div style="border: 1px solid black; padding: 2px; display: flex; justify-content: space-between; align-items: center;"> 1 1 1 0 R R R R 0 1 1 i i i i i </div>

Instruction Fields

Rx	RRRR	AGU Source/Destination Register					
0000	N0	0100	—	1000	R0	1100	R4
0001	N1	0101	—	1001	R1	1101	R5
0010	N2	0110	—	1010	R2	1110	R6
0011	N3	0111	SP	1011	R3	1111	R7

Note: This instruction can specify R8-R15 as operands by using a high register prefix.



#u5

iiii

5-bit unsigned immediate data = 1, set by the assembler



DECEQ Decrement and Set T If Equal Zero (DALU) DECEQ

Operation

Assembler Syntax

$D_n - 1 \rightarrow D_n$; if $D_n == 0$, then $1 \rightarrow T$, else $0 \rightarrow T$ `DECEQ Dn`

Description

DECEQ Dn

Decrements a data register (D_n) and sets the T bit if the result is equal to zero.

Status and Conditions that Affect Instruction

None.

Status and Conditions Changed by Instruction

Register Address	Bit Name	Description
SR[0]	C	Calculates and updates the carry bit in the status register.
SR[1]	T	Set if result = 0, cleared otherwise.
EMR[2]	DOVF	Set if the result cannot be represented in 40 bits.
L_n	L	Clears the L_n bit in the destination register.

Example

`deceq d7`

Register/Memory Address	Before	After
L7:D7	\$0:\$00 0000 0001	\$0:\$00 0000 0000
SR	\$00E4 0000	\$00E4 0002
EMR		\$0000 0000

Instruction Formats and Opcodes

Instruction	Words	Cycles	Type	Opcode
DECEQ D_n	1	1	1	<div style="display: flex; justify-content: space-between; align-items: center;"> 15 8 7 0 </div> <div style="border: 1px solid black; padding: 2px; display: flex; justify-content: space-between; align-items: center;"> 0 * 1 0 0 1 F F F 1 1 0 1 1 0 1 </div>

Note: ** indicates serial grouping encoding.



Instruction Fields

Dn

FFF

Single Source/Destination Data Register

000	D0	010	D2	100	D4	110	D6
001	D1	011	D3	101	D5	111	D7

Note: This instruction can specify D8-D15 as operands by using a prefix.



DECEQA Decrement and Set T If Equal Zero (AGU)

DECEQA

Operation

Assembler Syntax

$Rx - 1 \rightarrow Rx$; if $Rx == 0$, then $1 \rightarrow T$, else $0 \rightarrow T$ DECEQA Rx

Description

DECEQA Rx

Decrements an AGU register (Rx) and sets the T bit if the result is zero. SP cannot be used as an operand of this instruction.

Status and Conditions that Affect Instruction

None.

Status and Conditions Changed by Instruction

Register Address	Bit Name	Description
SR[1]	T	Set if result = 0, cleared otherwise.

Example

deceqa r0

Register/Memory Address	Before	After
R0	\$0000 0001	\$0000 0000
SR	\$00E4 0000	\$00E4 0002

Instruction Formats and Opcodes

Instruction	Words	Cycles	Type	Opcode
DECEQA Rx	1	1	2	<div style="display: flex; justify-content: space-between; align-items: center;"> 15 8 7 0 </div> <div style="border: 1px solid black; padding: 2px; display: flex; justify-content: space-between; align-items: center;"> 1 1 1 0 R R R R 1 1 1 1 0 1 1 0 </div>

Instruction Fields

Rx	RRRR	AGU Source/Destination Register						
	0000	N0	0100	—	1000	R0	1100	R4
	0001	N1	0101	—	1001	R1	1101	R5
	0010	N2	0110	—	1010	R2	1110	R6
	0011	N3	0111	SP	1011	R3	1111	R7

Note: This instruction can specify R8-R15 as operands by using a high register prefix. SP cannot be used as an operand for this instruction.

Operation

 $D_n - 1 \rightarrow D_n; D_n \geq 0 \rightarrow T$

Assembler Syntax

`DECGE Dn`

Description

DECGE Dn

Decrements a data register (Dn) and sets the T bit if the result is greater than or equal to zero. In the case of an arithmetic overflow (DECGE on the value \$80 0000 0000), the T bit will not be set.

Status and Conditions that Affect Instruction

None.

Status and Conditions Changed by Instruction

Register Address	Bit Name	Description
SR[0]	C	Calculates and updates the carry bit in the status register.
SR[1]	T	Set if result ≥ 0 , cleared otherwise.
EMR[2]	DOVF	Set if the result cannot be represented in 40 bits.
Ln	L	Clears the Ln bit in the destination register.

Example

```
decge
```

Instruction	SR	d3
<code>move.w #\$1,d3</code>	<code>;\$00E4 0000</code>	<code>\$00 0000 0001</code>
<code>decge d3</code>	<code>;\$00E4 0002</code>	<code>T-bit set \$00 0000 0000</code>
<code>decge d3</code>	<code>;\$00E4 0001</code>	<code>T-bit cleared, carry bit set \$FF FFFF FFFF</code>

Instruction Formats and Opcodes

Instruction	Words	Cycles	Type	Opcode
DECGE Dn	1	1	1	<div style="display: flex; justify-content: space-between; align-items: center;"> 15 8 7 0 </div> <div style="border: 1px solid black; padding: 2px; display: flex; justify-content: space-between; align-items: center;"> 0 * 1 0 0 1 F F F 1 1 0 1 1 0 0 </div>

Note: ** indicates serial grouping encoding.



struction Fields

Dn **FFF** **Single Source/Destination Data Register**

000	D0	010	D2	100	D4	110	D6
001	D1	011	D3	101	D5	111	D7

Note: This instruction can specify D8-D15 as operands by using a prefix.



DECGEA

Decrement and Set T
If Greater Than or Equal to Zero (AGU)

DECGEA

Operation

$Rx - 1 \rightarrow Rx$; $Rx \geq 0 \rightarrow T$

Assembler Syntax

DECGEA Rx

Description

DECGEA Rx

Decrements an AGU register (Rx) and sets the T bit if the result is greater than or equal to zero. In case there is an arithmetic overflow (DECGEA on the value of \$80000000), the T bit will not be set by this instruction. SP cannot be used as an operand of this instruction.

Status and Conditions that Affect Instruction

None.

Status and Conditions Changed by Instruction

Register Address	Bit Name	Description
SR[1]	T	Set if result ≥ 0 , cleared otherwise.

Example 1

decgea r4

Register/Memory Address	Before	After
R4	\$0010 E438	\$0010 E437
SR	\$00E4 0000	\$00E4 0002

Example 2

decgea r4

Register/Memory Address	Before	After
R4	\$8000 0000	\$7FFF FFFF
SR	\$00E4 0002	\$00E4 0000



Instruction Formats and Opcodes

Instruction	Words	Cycles	Type	Opcode
DECGEA Rx	1	1	2	<div style="display: flex; justify-content: space-between; align-items: center;"> 15 8 7 0 </div> <div style="border: 1px solid black; padding: 2px; display: flex; justify-content: space-between; align-items: center;"> 1 1 1 0 R R R R 1 1 1 1 0 1 1 1 </div>

Instruction Fields

Rx	RRRR	AGU Source/Destination Register					
	0000 N0	0100	—	1000	R0	1100	R4
	0001 N1	0101	—	1001	R1	1101	R5
	0010 N2	0110	—	1010	R2	1110	R6
	0011 N3	0111	SP	1011	R3	1111	R7

Note: This instruction can specify R8-R15 as operands by using a high register prefix. SP cannot be used by this instruction.

Operation

1 → DI

Assembler Syntax

DI

Description

DI

Sets the DI bit in the status register in order to disable interrupts. The effect is immediate, so the instructions that execute in the same execution set as well as later execution sets are not interruptible by maskable interrupts. Non-maskable interrupts and exceptions are not disabled by this bit.

The DI instruction and its counterpart, the EI instruction, can be used to delimit a code segment that needs to be protected from interruption. For example, a non-interruptible read-modify-write sequence of execution sets could be written like this:

```
DI read
modify
EI write
```

Where read, modify, and write stand for instruction(s). If using this instruction, no allowance is necessary for a pipeline delay of updating SR by the DI instruction. This instruction can appear only once in an execution set.

Status and Conditions that Affect Instruction

Register Address	Bit Name	Description
SR[18]	EXP	Determines execution working mode.

Status and Conditions Changed by Instruction

Register Address	Bit Name	Description
SR[19]	DI	Set disable interrupt bit.

Example 1

di

Register/Memory Address	Before	After
SR	\$0000 0000	\$0008 0000
EMR	\$0000 0000	\$0000 0000

Instruction Formats and Opcodes

Instruction	Words	Cycles	Type	Opcode
-------------	-------	--------	------	--------



DI 1 1 4

15	8	7	0													
1	0	0	1	1	1	1	1	1	0	1	1	1	1	1	0	1

Operation

If $Dn[39] \oplus Da[39] = 1$,
 then $2 * Dn + C + (Da \& \$FF\ FFFF\ 0000) \rightarrow Dn$
 else $2 * Dn + C - (Da \& \$FF\ FFFF\ 0000) \rightarrow Dn$
 where \oplus denotes the bitwise exclusive OR operator.

Assembler Syntax

DIV Da, Dn

Description

DIV Da, Dn

This instruction is used iteratively to divide the destination operand Dn by the source operand Da and store the result in the destination operand Dn. The 32-bit dividend must be a positive fraction which has been sign-extended to 40-bits and stored in the full 40-bit Dn. The 16-bit divisor is a signed fraction and is stored in Da.

Each DIV iteration calculates one quotient bit using a non-restoring fractional division algorithm (see description below). After the execution of the first DIV instruction, Dn holds both the partial remainder and the formed quotient. The partial remainder occupies the high portion of Dn and is a signed fraction. The formed quotient occupies the low portion of Dn and is a positive fraction. One bit of the formed quotient is shifted into bit 0 of Dn at the start of each DIV iteration. The formed quotient is the true quotient if the true quotient is positive. If the true quotient is negative, the formed quotient must be negated. Valid results are obtained only when $|Dn| < |Da|$ and the operands are interpreted as fractions. This condition ensures that the magnitude of the quotient is less than one (i.e., is fractional) and precludes division by zero.

The DIV instruction calculates one quotient bit based on the divisor and the previous partial remainder. To produce an N-bit quotient, the DIV instruction is executed N times, where N is the number of bits of precision desired in the quotient, $1 \leq N \leq 16$. Thus, for a full-precision (16 bit) quotient, 16 DIV iterations are required. In general, executing the DIV instruction N times produces an N-bit quotient and a 32-bit remainder that has (32-N) bits of precision and whose N most significant bits are zeros. The partial remainder is not a true remainder and must be corrected (due to the non-restoring nature of the division algorithm) before it can be used. Therefore, once the divide is complete, it is necessary to reverse the last DIV operation, and restore the remainder to obtain the true remainder.

The DIV instruction uses a non-restoring fractional division algorithm that consists of the following operations (see the previous Operation definition):

1. **Compare the source and destination operand sign bits:** An exclusive OR operation is performed on bit 39 of Dn and bit 39 of Da.
2. **Shift the partial remainder and the quotient:** Dn is shifted one bit to the left. The carry bit C is moved into bit 0 of Dn. The carry bit represents the quotient bit generated by the previous DIV iteration.

3. **Calculate the next quotient bit and the new partial remainder:** The 16-bit signed divisor in Da.H is either added to or subtracted from Dn.H, and the result is stored back into Dn.H. If the result of the exclusive OR operation previously described was a “1” (i.e., the sign bits were different), Da.H is added to Dn.H. If the result of the exclusive OR operation was a “0” (i.e., the sign bits were the same), Da.H is subtracted from Dn.H. Because of the sign-extension of the 16-bit signed divisor, the addition or subtraction operation correctly sets the carry bit C of the condition code register with the next quotient bit.

For extended precision division (i.e., for N-bit quotients where $N > 16$), the DIV instruction is no longer applicable, and a user-defined N-bit division routine is required.

For further information on division algorithms, refer to pages 524–530 of *Theory and Application of Digital Signal Processing* by Rabiner and Gold (Prentice-Hall, 1975), pages 190–199 of *Computer Architecture and Organization* by John Hayes (McGraw-Hill, 1978), pages 213–223 of *Computer Arithmetic: Principles, Architecture, and Design* by Kai Hwang (John Wiley and Sons, 1979), or other references as required.

Status and Conditions that Affect Instruction

Register Address	Bit Name	Description
SR[0]	C	Carry bit is copied into Dn[0].
SR[5:4]	S[1:0]	Scaling mode bits determine which bits in the result are used in the Ln bit calculation.

Status and Conditions Changed by Instruction

Register Address	Bit Name	Description
SR[0]	C	Bit is set if bit 39 of the result is cleared.
EMR[2]	DOVF	Set if the MS bit of the result cannot be represented in 40 bits, or is changed as a result of the instruction's left shift operation.
Ln	L	Calculates and updates the Ln bit in the destination register.

Example

```
div d2,d1
```

Register/Memory Address	Before	After
D2	\$00 2311 5A3B	
L1:D1	\$0:\$00 6666 0A57	\$0:\$00 A9BB 14AE
SR	\$00E4 0000	\$00E4 0001
EMR		\$0000 0000

Instruction Formats and Opcodes

Instruction	Words	Cycles	Type	Opcode																
DIV Da, Dn	1	1	1	<div style="display: flex; justify-content: space-between; align-items: center;"> 15 8 7 0 </div> <table border="1" style="width: 100%; text-align: center;"> <tr> <td>0</td><td>*</td><td>1</td><td>1</td><td>0</td><td>0</td><td>F</td><td>F</td><td>F</td><td>1</td><td>0</td><td>1</td><td>0</td><td>J</td><td>J</td><td>J</td> </tr> </table>	0	*	1	1	0	0	F	F	F	1	0	1	0	J	J	J
0	*	1	1	0	0	F	F	F	1	0	1	0	J	J	J					

Note: ** indicates serial grouping encoding.

Instruction Fields

Dn **FFF** **Single Source/Destination Data Register**

000	D0	010	D2	100	D4	110	D6
001	D1	011	D3	101	D5	111	D7

Note: This instruction can specify D8-D15 as operands by using a prefix.

Da **JJJ** **Single Source Data Register**

000	D0	010	D2	100	D4	110	D6
001	D1	011	D3	101	D5	111	D7

Note: This instruction can specify D8-D15 as operands by using a prefix.



DMACSS Multiply Signed By Signed and Accumulate With Right Shifted Data Register (DALU) DMACSS

Operation

$[Dn \gg 16] + Dc.H * Dd.H \rightarrow Dn$
(Dc signed, Dd signed)

Assembler Syntax

DMACSS Dc, Dd, Dn

Description

DMACSS Dc,Dd,Dn

Shifts Dn 16 bits to the right with bit 39 sign-extended into bits [39:24]. Adds the result to the product of signed fractions in Dc.H and Dd.H. Places the result into Dn.

Dc and Dd are a data register pair. The operands are in the HP of each register.

This instruction is optimized for multi-precision-multiplication support.

Status and Conditions that Affect Instruction

None.

Status and Conditions Changed by Instruction

Register Address	Bit Name	Description
Ln	L	Clears the Ln bit in the destination register.

Example

```
dmacss d2, d3, d5
```

Register/Memory Address	Before	After
D2	\$00 0002 0000	
D3	\$00 0003 0000	
L5:D5	\$0:\$00 0050 0000	\$0:\$00 0000 005C
EMR		\$0000 0000

\$00 0002 0000	2^{-14}
x \$00 0003 0000	$2^{-14} + 2^{-15}$
<hr/>	
\$00 0000 000C	$2^{-28} + 2^{-29}$
+ \$00 0000 0050	
<hr/>	
\$00 0000 005C	

Instruction Formats and Opcodes

Instruction	Words	Cycles	Type	Opcode																	
DMACSS	Dc, Dd, Dn	1	1	1	<div style="display: flex; justify-content: space-between; align-items: center;"> 15 8 7 0 </div> <table border="1" style="width: 100%; text-align: center;"> <tr> <td>0</td><td>*</td><td>1</td><td>0</td><td>1</td><td>1</td><td>F</td><td>F</td><td>F</td><td>1</td><td>1</td><td>1</td><td>0</td><td>1</td><td>e</td><td>e</td> </tr> </table>	0	*	1	0	1	1	F	F	F	1	1	1	0	1	e	e
0	*	1	0	1	1	F	F	F	1	1	1	0	1	e	e						

Note: ** indicates serial grouping encoding.

Instruction Fields

Dc, Dd **ee** **Data Register Pairs**

00	D0, D1	01	D2, D3	10	D4, D5	11	D6, D7
----	--------	----	--------	----	--------	----	--------

Note: This instruction can specify D8-D15 as operands by using a prefix.

Dn **FFF** **Single Source/Destination Data Register**

000	D0	010	D2	100	D4	110	D6
001	D1	011	D3	101	D5	111	D7

Note: This instruction can specify D8-D15 as operands by using a prefix.

struction Fields

Dc,Dd
ee
Data Register Pairs

00	D0,D1	01	D2,D3	10	D4,D5	11	D6,D7
----	-------	----	-------	----	-------	----	-------

Note: This instruction can specify D8-D15 as operands by using a prefix.

Dn
FFF
Single Source/Destination Data Register

000	D0	010	D2	100	D4	110	D6
001	D1	011	D3	101	D5	111	D7

Note: This instruction can specify D8-D15 as operands by using a prefix.



DOENn

DO Enable Long Loop (AGU)

DOENn

Operation

#u6 → LCn; 1 → LFn

#u16 → LCn; 1 → LFn

DR → LCn; 1 → LFn

Assembler Syntax

DOENn #u6 { $0 \leq u6 < 64$ }

DOENn #u16 { $0 \leq u16 < 2^{16}$ }

DOENn DR

Description

This instruction initializes the selected loop as a long loop by loading the iteration count into the respective loop counter and setting the respective loop flag in the SR. After this instruction is executed, the loop becomes active. There can be other instructions between this instruction and the actual body of the loop. If the loop is nested, the DOEN instruction must be placed inside the enclosing loop in order to re-activate the inner loop each iteration. Various programming rules apply concerning the minimum distance between this instruction and the loop body or other loop instructions.

DOENn #u6

Moves an unsigned 6-bit immediate value into the loop counter (LCn) and enables the chosen loop flag.

DOENn #u16

Moves an unsigned 16-bit immediate value into the loop counter (LCn) and enables the chosen loop flag.

DOENn DR

Moves the 32 lower bits of data or address register into the loop counter (LCn) and enables the chosen loop flag.

Status and Conditions that Affect Instruction

None.

Status and Conditions Changed by Instruction

Register Address	Bit Name	Description
SR[30:27]	LF[3:0]	Sets active loopflag.

Example

doen2 d0

Register/Memory Address	Before	After
D0	\$00 0000 000F	
LC2		\$0000 000F
SR	\$00E4 0000	\$20E4 0000



DOENSHn Do Enable Short Loop (AGU) DOENSHn

Operation

#u6 → LCn; 1 → LFn; 1 → SLF

#u16 → LCn; 1 → LFn; 1 → SLF

DR → LCn; 1 → LFn; 1 → SLF

Assembler Syntax

DOENSHn #u6 {0 ≤ u6 < 64}

DOENSHn #u16 {0 ≤ u16 < 2¹⁶}

DOENSHn DR

Description

This instruction initializes the selected loop as a short loop by loading the iteration count to the respective loop counter and setting the SLF and respective loop flag in the SR. After this instruction is executed, the loop becomes active. There can be a distance between this instruction and the actual body of the loop. In case the loop is nested, the DOENSH instruction must be placed inside the enveloping loop in order to re-activate the inner loop each iteration. Various programming rules apply concerning the minimum distance between this instruction and the loop body or other loop instructions.

DOENSHn #u6

Moves an unsigned 6-bit immediate value into the loop counter (LCn) and enables the chosen loop flag and short loop flag.

DOENSHn #u16

Moves an unsigned 16-bit immediate value into the loop counter (LCn) and enables the chosen loop flag and short loop flag.

DOENSHn DR

Moves the 32 lower bits of a data or address register into the loop counter (LCn) and enables the chosen loop flag and short loop flag.

Status and Conditions that Affect Instruction

None.

Status and Conditions Changed by Instruction

Register Address	Bit Name	Description
SR[30:27]	LF[3:0]	Sets active loopflag.
SR[31]	SLF	Sets short loopflag.

Example

doensh2 d0

Register/Memory Address	Before	After
D0	\$00 0000 000F	
LC2		\$0000 000F



DOSETUPn

Setup Long Loop
Start Address (AGU)

DOSETUPn

Operation

PC + displacement → SAn

Assembler Syntax

DOSETUPn label

Description

DOSETUPn label

This instruction is required for initialization of a long loop, not short loops. In case the loop is nested, the DOSETUPn instruction can be placed outside the enveloping loop as long as SA (Start Address) is not changed by instructions in the loop. DOSETUPn loads a loop start address register (SAn). The label is placed at the beginning of the loop. The encoded value in the DOSETUP instruction is a PC relative displacement calculated by the assembler and linker from the label. The start address placed in SAn is the absolute address of the label. The DOSETUPn instruction is redundant with the LOOPSTART assembler directive, both of which define SA. In case of a conflict between the two, SA is defined by DOSETUPn.

Status and Conditions that Affect Instruction

None.

Status and Conditions Changed by Instruction

None.

Example

```
dosetupl _label
```

Register/Memory Address	Before	After
(displacement)	\$101E	
PC	\$0000 0002	
SA1		\$0000 1020



Instruction Formats and Opcodes

Instruction	Words	Cycles	Type	Opcode																																
DOSETUPn label	2	1	4	<div style="display: flex; justify-content: space-between; width: 100%;"> 15 8 7 0 </div> <table border="1" style="width: 100%; border-collapse: collapse;"> <tr> <td style="width: 10%;">0</td><td style="width: 10%;">0</td><td style="width: 10%;">1</td><td style="width: 10%;">0</td><td style="width: 10%;">1</td><td style="width: 10%;">0</td><td style="width: 10%;">n</td><td style="width: 10%;">n</td><td style="width: 10%;">A</td><td style="width: 10%;">A</td><td style="width: 10%;">A</td><td style="width: 10%;">0</td><td style="width: 10%;">0</td><td style="width: 10%;">0</td><td style="width: 10%;">1</td><td style="width: 10%;">1</td> </tr> <tr> <td>1</td><td>0</td><td>0</td><td>A</td><td>A</td><td>A</td><td>A</td><td>A</td><td>A</td><td>A</td><td>A</td><td>A</td><td>A</td><td>A</td><td>A</td><td>a</td> </tr> </table>	0	0	1	0	1	0	n	n	A	A	A	0	0	0	1	1	1	0	0	A	A	A	A	A	A	A	A	A	A	A	A	a
0	0	1	0	1	0	n	n	A	A	A	0	0	0	1	1																					
1	0	0	A	A	A	A	A	A	A	A	A	A	A	A	a																					

Instruction Fields

n	Loop Identifier						
00	Loop 0	01	Loop 1	10	Loop 2	11	Loop 3

displacement aAAAAAAAAAAAAAAAAA0 16-bit signed PC relative displacement.
 The encoding is the displacement with bit 0 stripped and replaced by the sign bit.



Enable Interrupts (AGU)



Operation

0 → DI

Assembler Syntax

EI

Description

EI

Clears the DI bit in the status register to enable interrupts. The EI instruction and its counterpart, the DI instruction, can be used to delimit a non-interruptible code sequence. For example, a non-interruptible read-modify-write sequence of execution sets can be written like this:

```
DI read
modify
EI write
```

Where read, modify, and write represent instruction(s). This instruction can appear only once in an execution set. The effect of EI may not be immediate. That is, a pending interrupt may not be serviced as the first execution set immediately after this instruction because of pipeline effects.

Status and Conditions that Affect Instruction

Register Address	Bit Name	Description
SR[18]	EXP	Determines execution working mode.

Status and Conditions Changed by Instruction

Register Address	Bit Name	Description
SR[19]	DI	Clears disable interrupt bit.

Example

```
ei
```

Register/Memory Address	Before	After
SR	\$EC0000	\$E40000

Instruction Formats and Opcodes

Instruction	Words	Cycles	Type	Opcode
EI	1	1	4	<div style="display: flex; justify-content: space-between; align-items: center;"> 15 8 7 0 </div> <div style="border: 1px solid black; padding: 2px; display: flex; justify-content: space-between;"> 1 0 0 1 1 1 1 1 0 1 1 1 1 1 0 0 </div>



EOR

Bitwise Exclusive OR (DALU)

EOR

Operation

$Da \oplus Dn \rightarrow Dn$

Assembler Syntax

EOR Da, Dn

Description

EOR Da,Dn

Performs a bitwise exclusive OR between two data registers (Da and Dn) and stores the result in a destination data register (Dn).

Status and Conditions that Affect Instruction

None.

Status and Conditions Changed by Instruction

Register Address	Bit Name	Description
Ln	L	Clears the Ln bit in the destination register.

Example

eor d4,d5

Register/Memory Address	Before	After
D4	\$FF FFFF FFFB	
L5:D5	\$0:\$00 0000 0003	\$0:\$FF FFFF FFF8

B	1011
$\oplus 3$	<u>0011</u>
8	1000

Instruction Formats and Opcodes

Instruction	Words	Cycles	Type	Opcode																
EOR Da, Dn	1	1	2	<div style="display: flex; justify-content: space-between; align-items: center;"> 15 8 7 0 </div> <table border="1" style="width: 100%; text-align: center; border-collapse: collapse;"> <tr> <td style="width: 10%;">1</td> <td style="width: 10%;">1</td> <td style="width: 10%;">0</td> <td style="width: 10%;">1</td> <td style="width: 10%;">1</td> <td style="width: 10%;">1</td> <td style="width: 10%;">F</td> <td style="width: 10%;">F</td> <td style="width: 10%;">F</td> <td style="width: 10%;">0</td> <td style="width: 10%;">0</td> <td style="width: 10%;">1</td> <td style="width: 10%;">0</td> <td style="width: 10%;">J</td> <td style="width: 10%;">J</td> <td style="width: 10%;">J</td> </tr> </table>	1	1	0	1	1	1	F	F	F	0	0	1	0	J	J	J
1	1	0	1	1	1	F	F	F	0	0	1	0	J	J	J					

Instruction Fields

Da **JJJ** **Single Source Data Register**

000	D0	010	D2	100	D4	110	D6
001	D1	011	D3	101	D5	111	D7

Note: This instruction can specify D8-D15 as operands by using a prefix.

Dn **FFF** **Single Source/Destination Data Register**

000	D0	010	D2	100	D4	110	D6
001	D1	011	D3	101	D5	111	D7

Note: This instruction can specify D8-D15 as operands by using a prefix.

EOR Bitwise Exclusive OR on a 16-Bit Operand (BMU) EOR

Operation

$\#u16 \oplus DR.L \rightarrow DR.L$

$\#u16 \oplus DR.H \rightarrow DR.H$

Assembler Syntax

EOR #u16,DR.L { $0 \leq u16 < 2^{16}$ }

EOR #u16,DR.H { $0 \leq u16 < 2^{16}$ }

Description

EOR #u16,DR.L

Performs a bitwise exclusive OR between a 16-bit unsigned immediate value and the LP of an address register or data register (DR). Stores the result in the destination register (DR). This instruction is assembler-mapped to BMCHG #u16,DR.L with the immediate value. The HP of the register is unaffected.

EOR #u16,DR.H

Performs a bitwise exclusive OR between a 16-bit unsigned immediate value and the HP of an address register or data register (DR). Stores the result in the destination register (DR). This instruction is assembler-mapped to BMCHG #u16,DR.H with the immediate value. The LP of the register is unaffected.

Status and Conditions that Affect Instruction

None.

Status and Conditions Changed by Instruction

Register Address	Bit Name	Description
Ln	L	Clears the Ln bit in the destination register.

Example

```
eor #$5,d5.l
```

Register/Memory Address	Before	After
immediate	\$0005	
d5	\$0000 0003	\$0000 0006

5	0101
$\oplus 3$	0011
6	0110

Instruction Formats and Opcodes

Instruction	Words	Cycles	Type	Opcode												
EOR #u16,DR.L	2	2	3	<table border="1"> <tr> <td style="text-align: right;">15</td> <td style="text-align: center;">8</td> <td style="text-align: center;">7</td> <td style="text-align: right;">0</td> </tr> <tr> <td>0 0 0 0 1 0 1 0</td> <td>i</td> <td>i</td> <td>i 0 H H H H</td> </tr> <tr> <td>1 0 1 i i i i i</td> <td>i</td> <td>i</td> <td>i i i i i</td> </tr> </table>	15	8	7	0	0 0 0 0 1 0 1 0	i	i	i 0 H H H H	1 0 1 i i i i i	i	i	i i i i i
15	8	7	0													
0 0 0 0 1 0 1 0	i	i	i 0 H H H H													
1 0 1 i i i i i	i	i	i i i i i													
EOR #u16,DR.H	2	2	3	<table border="1"> <tr> <td style="text-align: right;">15</td> <td style="text-align: center;">8</td> <td style="text-align: center;">7</td> <td style="text-align: right;">0</td> </tr> <tr> <td>0 0 0 0 1 0 1 0</td> <td>i</td> <td>i</td> <td>i 1 H H H H</td> </tr> <tr> <td>1 0 1 i i i i i</td> <td>i</td> <td>i</td> <td>i i i i i</td> </tr> </table>	15	8	7	0	0 0 0 0 1 0 1 0	i	i	i 1 H H H H	1 0 1 i i i i i	i	i	i i i i i
15	8	7	0													
0 0 0 0 1 0 1 0	i	i	i 1 H H H H													
1 0 1 i i i i i	i	i	i i i i i													

Instruction Fields

DR	HHHH	Data/Address Register					
0000	D0	0100	D4	1000	R0	1100	R4
0001	D1	0101	D5	1001	R1	1101	R5
0010	D2	0110	D6	1010	R2	1110	R6
0011	D3	0111	D7	1011	R3	1111	R7

Note: This instruction can specify D8-D15 or R8-R15 as operands by using a high register prefix.

#u16 `iiiiiiiiiiiiiiii` 16-bit unsigned immediate data



EOR.W

Bitwise Exclusive OR on a 16-Bit Operand in Memory (BMU)

EOR.W

Operation

$\#u16 \oplus (R) \rightarrow (R)$

EOR.W $\#u16, (Rn) \{0 \leq u16 < 2^{16}\}$

$\#u16 \oplus (SP-u5) \rightarrow (SP-u5)$

EOR.W $\#u16, (SP-u5) \{0 \leq u16 < 2^{16}\} \{0 \leq u5 < 64, W\}$

$\#u16 \oplus (SP+s16) \rightarrow (SP+s16)$

EOR.W $\#u16, (SP+s16) \{0 \leq u16 < 2^{16}\} \{-2^{15} \leq s16 < 2^{15}, W\}$

$\#u16 \oplus (a16) \rightarrow (a16)$

EOR.W $\#u16, (a16) \{0 \leq u16 < 2^{16}\} \{0 \leq a16 < 2^{16}, W\}$

Description

These operations read from memory, modify the retrieved value, and write the new value back to the same memory address, resulting in two memory accesses. The absolute addresses, offsets, and address register values must be word-aligned.

EOR.W $\#u16, (Rn)$

Performs a bitwise exclusive OR between an immediate unsigned word and the contents of a memory address, pointed to by the contents of an address register (Rn). Stores the result in the same memory address. This instruction is assembler-mapped to BMCHG.W $\#u16, (Rn)$ with the immediate value.

EOR.W $\#u16, (SP-u5)$

Performs a bitwise exclusive OR between an immediate unsigned word and the contents of a memory address. Stores the result in the same memory address. The memory address is calculated as the active stack pointer (SP) minus a 5-bit unsigned offset value. This instruction is assembler-mapped to BMCHG.W $\#u16, (SP-u5)$ with the immediate value.

EOR.W $\#u16, (SP+s16)$

Performs a bitwise exclusive OR between an immediate unsigned word and the contents of a memory address. Stores the result in the same memory address. The memory address is calculated as the active stack pointer (SP) plus a 16-bit signed offset value. This instruction is assembler-mapped to BMCHG.W $\#u16, (SP+s16)$ with the immediate value.

EOR.W $\#u16, (a16)$

Performs a bitwise exclusive OR between an immediate unsigned word and the contents of an absolute memory address. Stores the result in the same memory address. This instruction is assembler-mapped to BMCHG.W $\#u16, (a16)$ with the immediate value.

Status and Conditions that Affect Instruction

Register Address	Bit Name	Description
SR[18]	EXP	Determines which stack pointer is used when the stack pointer is an operand. Otherwise, the instruction is not affected by SR.

Status and Conditions Changed by Instruction

None.



Example

eor.w #0xaaaa, (r0)

Register/Memory Address	Before	After
immediate	\$AAAA	
(r0)	\$0000 5555	\$0000 FFFF

Instruction Formats and Opcodes

Instruction	Words	Cycles	Type	Opcode																																																																
EOR.W #u16, (Rn)	2	2	3	<table border="1" style="width: 100%; text-align: center;"> <tr> <td colspan="4">15</td> <td colspan="4">8</td> <td colspan="4">7</td> <td colspan="4">0</td> </tr> <tr> <td>0</td><td>0</td><td>0</td><td>1</td><td>0</td><td>0</td><td>1</td><td>0</td> <td>i</td><td>i</td><td>i</td><td>0</td><td>1</td><td>R</td><td>R</td><td>R</td> </tr> <tr> <td>1</td><td>0</td><td>1</td><td>i</td><td>i</td><td>i</td><td>i</td><td>i</td> <td>i</td><td>i</td><td>i</td><td>i</td><td>i</td><td>i</td><td>i</td><td>i</td> </tr> </table>	15				8				7				0				0	0	0	1	0	0	1	0	i	i	i	0	1	R	R	R	1	0	1	i	i	i	i	i	i	i	i	i	i	i	i	i																
15				8				7				0																																																								
0	0	0	1	0	0	1	0	i	i	i	0	1	R	R	R																																																					
1	0	1	i	i	i	i	i	i	i	i	i	i	i	i	i																																																					
EOR.W #u16, (SP-u5)	2	3	3	<table border="1" style="width: 100%; text-align: center;"> <tr> <td colspan="4">15</td> <td colspan="4">8</td> <td colspan="4">7</td> <td colspan="4">0</td> </tr> <tr> <td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>1</td><td>0</td> <td>i</td><td>i</td><td>i</td><td>A</td><td>A</td><td>A</td><td>A</td><td>A</td> </tr> <tr> <td>1</td><td>0</td><td>1</td><td>i</td><td>i</td><td>i</td><td>i</td><td>i</td> <td>i</td><td>i</td><td>i</td><td>i</td><td>i</td><td>i</td><td>i</td><td>i</td> </tr> </table>	15				8				7				0				0	0	0	0	0	0	1	0	i	i	i	A	A	A	A	A	1	0	1	i	i	i	i	i	i	i	i	i	i	i	i	i																
15				8				7				0																																																								
0	0	0	0	0	0	1	0	i	i	i	A	A	A	A	A																																																					
1	0	1	i	i	i	i	i	i	i	i	i	i	i	i	i																																																					
EOR.W #u16, (SP+s16)	3	3	3	<table border="1" style="width: 100%; text-align: center;"> <tr> <td colspan="4">15</td> <td colspan="4">8</td> <td colspan="4">7</td> <td colspan="4">0</td> </tr> <tr> <td>0</td><td>0</td><td>1</td><td>1</td><td>1</td><td>0</td><td>1</td><td>0</td> <td>A</td><td>A</td><td>A</td><td>i</td><td>i</td><td>0</td><td>1</td><td>1</td> </tr> <tr> <td>0</td><td>0</td><td>1</td><td>A</td><td>A</td><td>A</td><td>A</td><td>A</td> <td>A</td><td>A</td><td>A</td><td>A</td><td>A</td><td>A</td><td>A</td><td>A</td> </tr> <tr> <td>1</td><td>0</td><td>i</td><td>i</td><td>i</td><td>i</td><td>i</td><td>i</td> <td>i</td><td>i</td><td>i</td><td>i</td><td>i</td><td>i</td><td>i</td><td>i</td> </tr> </table>	15				8				7				0				0	0	1	1	1	0	1	0	A	A	A	i	i	0	1	1	0	0	1	A	A	A	A	A	A	A	A	A	A	A	A	A	1	0	i	i	i	i	i	i	i	i	i	i	i	i	i	i
15				8				7				0																																																								
0	0	1	1	1	0	1	0	A	A	A	i	i	0	1	1																																																					
0	0	1	A	A	A	A	A	A	A	A	A	A	A	A	A																																																					
1	0	i	i	i	i	i	i	i	i	i	i	i	i	i	i																																																					
EOR.W #u16, (a16)	3	2	3	<table border="1" style="width: 100%; text-align: center;"> <tr> <td colspan="4">15</td> <td colspan="4">8</td> <td colspan="4">7</td> <td colspan="4">0</td> </tr> <tr> <td>0</td><td>0</td><td>1</td><td>1</td><td>1</td><td>0</td><td>1</td><td>0</td> <td>A</td><td>A</td><td>A</td><td>i</td><td>i</td><td>0</td><td>0</td><td>1</td> </tr> <tr> <td>0</td><td>0</td><td>1</td><td>A</td><td>A</td><td>A</td><td>A</td><td>A</td> <td>A</td><td>A</td><td>A</td><td>A</td><td>A</td><td>A</td><td>A</td><td>A</td> </tr> <tr> <td>1</td><td>0</td><td>i</td><td>i</td><td>i</td><td>i</td><td>i</td><td>i</td> <td>i</td><td>i</td><td>i</td><td>i</td><td>i</td><td>i</td><td>i</td><td>i</td> </tr> </table>	15				8				7				0				0	0	1	1	1	0	1	0	A	A	A	i	i	0	0	1	0	0	1	A	A	A	A	A	A	A	A	A	A	A	A	A	1	0	i	i	i	i	i	i	i	i	i	i	i	i	i	i
15				8				7				0																																																								
0	0	1	1	1	0	1	0	A	A	A	i	i	0	0	1																																																					
0	0	1	A	A	A	A	A	A	A	A	A	A	A	A	A																																																					
1	0	i	i	i	i	i	i	i	i	i	i	i	i	i	i																																																					

Instruction Fields

Rn	RRR	Address Register						
	000	R0	010	R2	100	R4	110	R6
	001	R1	011	R3	101	R5	111	R7

Note: This instruction can specify R8-R15 as operands by using a high register prefix.

- a16** AAAAAAAAAAAAAAAAAA 16-bit unsigned absolute address
- #u16** iiiiiiiiiiiiii 16-bit unsigned immediate data
- u5** AAAAA0 5-bit unsigned SP address offset
- s16** AAAAAAAAAAAAAAAAAA 16-bit signed SP address offset

EXTRACT Extract Signed Bit Field (DALU) EXTRACT

Operation

Db[(offset + width - 1):offset] → Dn[(width - 1):0]
 Db[offset + width - 1] → Dn[39:width] (sign-extension)

width = #U6; offset = #u6

width = Da[13:8]; offset = Da[5:0]

Assembler Syntax

```
EXTRACT #U6,#u6,Db,Dn
{0 ≤ U6 ≤ 40}{0 ≤ u6 ≤ 40}
{#U6+#u6 ≤ 40}
```

```
EXTRACT Da,Db,Dn
{0 ≤ Da[13:8] ≤ 40}
{0 ≤ Da[5:0] ≤ 40}
{Da[13:8]+Da[5:0] ≤ 40}
```

Description

These operations extract a bit field from a source data register (Db) and place it in a destination data register (Dn), right-aligned and sign-extended from the MSB of the extracted bit field. The extracted field is a signed integer. If the offset is zero, this instruction can be used to sign-extend an arbitrary width signed integer.

EXTRACT #U6,#u6,Db,Dn

Uses two immediate unsigned 6-bit integers for the width (#U6) and offset (#u6).

EXTRACT Da,Db,Dn

Uses a supplemental data register (Da) for the width (bits 13–8) and the offset (bits 5–0).

Status and Conditions that Affect Instruction

None.

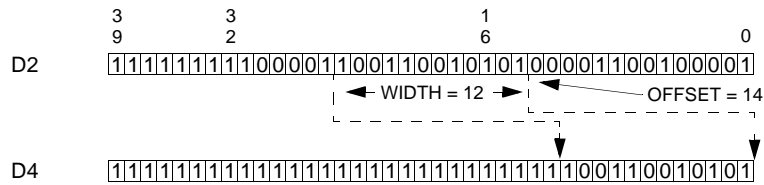
Status and Conditions Changed by Instruction

Register Address	Bit Name	Description
Ln	L	Clears the Ln bit in the destination register.

Example

```
extract #$c,#$e,d2,d4
```

Register/Memory Address	Before	After
immediate (width)	<input type="text" value="\$C"/>	
immediate (offset)	<input type="text" value="\$E"/>	
D2	<input type="text" value="\$FF 8665 4321"/>	
L4:D4	<input type="text" value="\$0:\$00 0000 0000"/>	<input type="text" value="\$0:\$FF FFFF F995"/>



Instruction Formats and Opcodes

Instruction	Words	Cycles	Type	Opcode												
EXTRACT #U6, #u6, Db, Dn	2	1	4	<table border="1"> <tr> <td>15</td> <td>8</td> <td>7</td> <td>0</td> </tr> <tr> <td>0 0 1 1 j j j 0</td> <td>1 1 0 0 1 F F F</td> <td></td> <td></td> </tr> <tr> <td>1 0 0 1 l l l l</td> <td>l l i i i i i i</td> <td></td> <td></td> </tr> </table>	15	8	7	0	0 0 1 1 j j j 0	1 1 0 0 1 F F F			1 0 0 1 l l l l	l l i i i i i i		
15	8	7	0													
0 0 1 1 j j j 0	1 1 0 0 1 F F F															
1 0 0 1 l l l l	l l i i i i i i															
EXTRACT Da, Db, Dn	2	1	4	<table border="1"> <tr> <td>15</td> <td>8</td> <td>7</td> <td>0</td> </tr> <tr> <td>0 0 1 1 j j j 0</td> <td>1 0 0 0 1 F F F</td> <td></td> <td></td> </tr> <tr> <td>1 0 0 1 0 0 0 0</td> <td>0 0 0 0 0 J J J</td> <td></td> <td></td> </tr> </table>	15	8	7	0	0 0 1 1 j j j 0	1 0 0 0 1 F F F			1 0 0 1 0 0 0 0	0 0 0 0 0 J J J		
15	8	7	0													
0 0 1 1 j j j 0	1 0 0 0 1 F F F															
1 0 0 1 0 0 0 0	0 0 0 0 0 J J J															

Instruction Fields

Da **JJJ** **Single Source Data Register**

000	D0	010	D2	100	D4	110	D6
001	D1	011	D3	101	D5	111	D7

Note: This instruction can specify D8-D15 as operands by using a prefix.

Dn **FFF** **Single Source/Destination Data Register**

000	D0	010	D2	100	D4	110	D6
001	D1	011	D3	101	D5	111	D7

Note: This instruction can specify D8-D15 as operands by using a prefix.

Db **jjj** **Single Source/Destination Data Register**

000	D0	010	D2	100	D4	110	D6
001	D1	011	D3	101	D5	111	D7

Note: This instruction can specify D8-D15 as operands by using a prefix.

#u16 iiiiii Unsigned 6-bit integer

#U16 IIIIII Unsigned 6-bit integer



EXTRACTU

Extract Unsigned Bit Field
(DALU)

EXTRACTU

Operation

$Db[(offset + width - 1):offset] \rightarrow Dn[(width - 1):0]$
 $0 \rightarrow Dn[39:width]$

width = #U6; offset = #u6

width = Da[13:8]; offset = Da[5:0]

Assembler Syntax

```
EXTRACTU #U6, #u6, Db, Dn
{ 0 ≤ U6 ≤ 40 }
{ 0 ≤ u6 ≤ 40 }
{ #U6 + #u6 ≤ 40 }
```

```
EXTRACTU Da, Db, Dn
{ 0 ≤ Da[13:8] ≤ 40 }
{ 0 ≤ Da[5:0] ≤ 40 }
{ Da[13:8] + Da[5:0] ≤ 40 }
```

Description

These operations extract a bit field from a source data register (Db) and place it in a destination data register (Dn), right-aligned and zero-extended. The extracted field is an unsigned integer. If the offset is zero, this instruction can be used to zero-extend an arbitrary width unsigned integer.

EXTRACTU #U6,#u6,Db,Dn

Uses two immediate unsigned 6-bit integers for the width (#U6) and offset (#u6).

EXTRACTU Da,Db,Dn

Uses a supplemental data register (Da) for the width (bits 13:8) and the offset (5:0).

Status and Conditions that Affect Instruction

None.

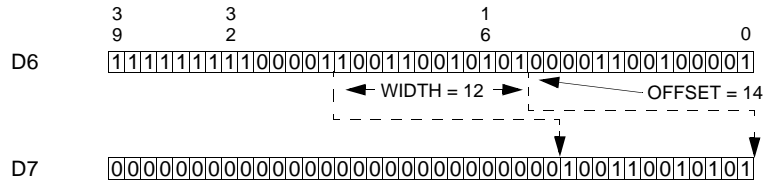
Status and Conditions Changed by Instruction

Register Address	Bit Name	Description
Ln	L	Clears the Ln bit in the destination register.

Example

```
extractu #c, #e, d2, d4
```

Register/Memory Address	Before	After
immediate (width)	<input type="text" value="\$C"/>	
immediate (offset)	<input type="text" value="\$E"/>	
D2	<input type="text" value="\$FF 8665 4321"/>	
L4:D4	<input type="text" value="\$0:\$00 0000 0000"/>	<input type="text" value="\$0:\$00 0000 0995"/>



Instruction Formats and Opcodes

Instruction	Words	Cycles	Type	Opcode												
EXTRACTU #U6, #u6, Da, Dn	2	1	4	<table border="1"> <tr> <td>15</td> <td>8</td> <td>7</td> <td>0</td> </tr> <tr> <td>0 0 1 1 j j j 0</td> <td>1 1 0 0 1 F F F</td> <td></td> <td></td> </tr> <tr> <td>1 0 0 0 l l l l</td> <td>l l i i i i i i</td> <td></td> <td></td> </tr> </table>	15	8	7	0	0 0 1 1 j j j 0	1 1 0 0 1 F F F			1 0 0 0 l l l l	l l i i i i i i		
15	8	7	0													
0 0 1 1 j j j 0	1 1 0 0 1 F F F															
1 0 0 0 l l l l	l l i i i i i i															
EXTRACTU Da, Db, Dn	2	1	4	<table border="1"> <tr> <td>15</td> <td>8</td> <td>7</td> <td>0</td> </tr> <tr> <td>0 0 1 1 j j j 0</td> <td>1 0 0 0 1 F F F</td> <td></td> <td></td> </tr> <tr> <td>1 0 0 0 0 0 0 0</td> <td>0 0 0 0 0 J J J</td> <td></td> <td></td> </tr> </table>	15	8	7	0	0 0 1 1 j j j 0	1 0 0 0 1 F F F			1 0 0 0 0 0 0 0	0 0 0 0 0 J J J		
15	8	7	0													
0 0 1 1 j j j 0	1 0 0 0 1 F F F															
1 0 0 0 0 0 0 0	0 0 0 0 0 J J J															

Instruction Fields

Da	JJJ	Single Source Data Register					
000	D0	010	D2	100	D4	110	D6
001	D1	011	D3	101	D5	111	D7

Note: This instruction can specify D8-D15 as operands by using a prefix.

Dn	FFF	Single Source/Destination Data Register					
000	D0	010	D2	100	D4	110	D6
001	D1	011	D3	101	D5	111	D7

Note: This instruction can specify D8-D15 as operands by using a prefix.

Db	jjj	Single Source/Destination Data Register					
000	D0	010	D2	100	D4	110	D6
001	D1	011	D3	101	D5	111	D7

Note: This instruction can specify D8-D15 as operands by using a prefix.

#u16 iiiiii unsigned 6-bit integer

#U16 IIIIII unsigned 6-bit integer



.ADDNC.W

Integer Addition

IADDNC.W

Without Changing the Carry Bit
Not Affected by Saturation (DALU)

Description

IADDNC.W #s16,Dn

Sign-extends the 16-bit immediate value to 40 bits and adds it to the destination data register Dn. The result is not affected by the arithmetic saturation mode. The carry bit is not affected by this instruction. This instruction is an integer (non-saturating) version of ADDNC.W.

Status and Conditions that Affect Instruction

None.

Status and Conditions Changed By Instruction

Register Address	Bit Name	Description
EMR[2]	DOVF	Set if the result cannot be represented in 40 bits.
Ln	L	Clears the L bit in the destination register.

Example

```
iaddnc.w #s16,d2
```

Register/Memory Address	Before	After
L2:D2	\$1:\$FFFFFFCA3E	\$0:\$FFFFFF6A40
EMR		\$00000000

Instruction Formats and Opcodes

Instruction	Words	Cycles	Type	Opcode												
IADDNC.W #s16,Dn	2	1	4	<table border="1"> <tr> <td>15</td> <td>8</td> <td>7</td> <td>0</td> </tr> <tr> <td>0 0 1 1 1 0 1 0</td> <td>i i i 1 0 F F F</td> <td></td> <td></td> </tr> <tr> <td>1 0 0 i i i i i</td> <td>i i i i i i i i</td> <td></td> <td></td> </tr> </table>	15	8	7	0	0 0 1 1 1 0 1 0	i i i 1 0 F F F			1 0 0 i i i i i	i i i i i i i i		
15	8	7	0													
0 0 1 1 1 0 1 0	i i i 1 0 F F F															
1 0 0 i i i i i	i i i i i i i i															

Instruction Fields

Dn	FFF	Single Source/Destination Data Register						
	000	D0	010	D2	100	D4	110	D6
	001	D1	011	D3	101	D5	111	D7

Note: If registers D8–D15 are accessed instead of D0–D7, a prefix is used.

#s16 iiiiiiiiiiiiii 16-bit signed immediate data

Operation

If T == 0, then execute group/subgroup
else treat as NOP

If T == 1, then execute group/subgroup
else treat as NOP

execute group/subgroup unconditionally

Assembler Syntax

IFF group or subgroup of instructions

IFT group or subgroup of instructions

IFA group or subgroup of instructions

Description

These instructions add conditional control over a group or subgroup of instructions in a VLES.

IFF

Execute the group or subgroup if T is equal to zero (condition is false). If T is equal to one (condition is true), the group or subgroup is treated as a NOP. This instruction can be used in conjunction with IFT to form an if/else clause.

IFT

Execute the group or subgroup if T is equal to one (condition is true). If T is equal to zero (condition is false), the group or subgroup is treated as a NOP. This instruction can be used in conjunction with IFF to form an if/else clause.

IFA

Always execute the group or subgroup. This instruction may be used in conjunction with IFT or IFF to split a VLES group into conditional and unconditional subgroups, where IFA must be the last subgroup in the VLES.

The following combinations of these instructions can be used:

```
IFT group ; execute group if T is set
IFF group ; execute group if T is clear
IFA group ; execute group unconditionally
IFT subgroup1 IFA subgroup2 ; execute subgroup1 if T is set,
; execute subgroup2 unconditionally
IFF subgroup1 IFA subgroup2 ; execute subgroup1 if T is clear,
; execute subgroup2 unconditionally
IFT subgroup1 IFF subgroup2 ; execute subgroup1 if T is set,
; execute subgroup2 if T is clear
```

- Notes:**
1. The instructions in the subgroups can be conditional (e.g., TFRT, JF), which adds finer control.
 2. The “IFA group” is the same as not using IFC. That is, unconditional execution of the VLES.
 3. The detailed use of IFC is defined by [Section 7.2, “VLES Grouping Semantics,”](#) and programming rule G.P.7 in [Section 7.5.3, “Prefix Grouping Rules.”](#)



Status and Conditions that Affect Instruction

Register Address	Bit Name	Description
SR[1]	T	True bit

Status and Conditions Changed by Instruction

None.

Example

```
ift move.w #>$ffff,d0
```

Register/Memory Address	Before	After
SR	\$00E4 0002	\$00E4 0002
immediate	\$FFFF	
L0:D0		\$0:\$FF FFFF FFFF

Instruction Formats and Opcodes

Instruction	Words	Cycles	Type	Opcode																																				
IFc	1	1	4	<table border="1"> <tr> <td>15</td> <td>8</td> <td>7</td> <td>0</td> </tr> <tr> <td>1</td> <td>0</td> <td>0</td> <td>1</td> </tr> <tr> <td>a</td> <td>a</td> <td>a</td> <td>0</td> </tr> <tr> <td>1</td> <td>1</td> <td>0</td> <td>p</td> </tr> <tr> <td>j</td> <td>c</td> <td>c</td> <td>c</td> </tr> </table>	15	8	7	0	1	0	0	1	a	a	a	0	1	1	0	p	j	c	c	c																
15	8	7	0																																					
1	0	0	1																																					
a	a	a	0																																					
1	1	0	p																																					
j	c	c	c																																					
IFc	2	1	4	<table border="1"> <tr> <td>15</td> <td>8</td> <td>7</td> <td>0</td> </tr> <tr> <td>0</td> <td>0</td> <td>1</td> <td>1</td> </tr> <tr> <td>a</td> <td>a</td> <td>a</td> <td>0</td> </tr> <tr> <td>H</td> <td>t</td> <td>h</td> <td>p</td> </tr> <tr> <td>j</td> <td>c</td> <td>c</td> <td>c</td> </tr> <tr> <td>1</td> <td>0</td> <td>1</td> <td>b</td> </tr> <tr> <td>B</td> <td>e</td> <td>E</td> <td>T</td> </tr> <tr> <td>b</td> <td>B</td> <td>e</td> <td>E</td> </tr> <tr> <td>b</td> <td>B</td> <td>e</td> <td>E</td> </tr> </table>	15	8	7	0	0	0	1	1	a	a	a	0	H	t	h	p	j	c	c	c	1	0	1	b	B	e	E	T	b	B	e	E	b	B	e	E
15	8	7	0																																					
0	0	1	1																																					
a	a	a	0																																					
H	t	h	p																																					
j	c	c	c																																					
1	0	1	b																																					
B	e	E	T																																					
b	B	e	E																																					
b	B	e	E																																					

Note: These instructions are encoded into either a one-word or two-word prefix.

Instruction Fields

ccc: Conditional execution of the entire execution set

In the following table, true/false relates to the state of the T bit in SR: D0, D1, D2, and D3 are DALU instructions. A0 and A1 are AGU instructions. The numbers relate to the relative offset of the instruction from the beginning of the set, as encoded. For example, a full execution set might be D0, D1, D2, D3, A0, A1.

- 000—Unconditionally executed
- 001—If true (D0,D2,A0), if false (D1,D3,A1)
- 010—If true, all the set
- 011—If false, all the set
- 100—Reserved
- 101—Reserved
- 110—If true (D0,D2,A0), always (D1,D3,A1)
- 111—If false (D0,D2,A0), always (D1,D3,A1)



ILLEGAL

Generate an Illegal Exception Request (AGU)

ILLEGAL

Operation

upon service: PC → (ESP); SR → (ESP + 4);
 SP + 8 → SP; VBA[31:12]: illegal_vector → PC;
 1 → EXP
 111 → I[2:0]
 1 → ILIN
 0 → C
 0 → T
 00 → S[1:0]
 0 → SLF
 0000 → LF[3:0]

Assembler Syntax

```
ILLEGAL {illegal_vector = $080}
```

Description

ILLEGAL

Generates an imprecise non-maskable illegal exception request. The exact place in the execution flow that the request is serviced depends on the machine state. Imprecise means that the exception timing is not guaranteed, being asynchronous with the instruction execution. Users should not rely on any timing between the ILLEGAL instruction execution and the start of exception processing. In the most common case, the exception vector is executed after four more execution sets are executed following the illegal instruction. In other cases, it can be the set immediately after or delayed by another execution set. Thus, it should be realized that in the exception routine, the machine state cannot be reconstructed to the exact state before or after the ILLEGAL instruction is executed. It is possible, however, to know at which PC the request was raised by reading the PC_EXCP register in the EOnCE (see the EOnCE section for a description of this register).

The AGU sets the EXP bit in SR to switch the active stack pointer to the exception stack pointer. .

Status and Conditions that Affect Instruction

None.

Status and Conditions Changed by Instruction

Register Address	Bit Name	Description
SR[18]	EXP	Sets EXP to switch active stack pointer to exception stack pointer.
SR[23:21]	I[2:0]	Set interrupt priority level to 111.
EMR[0]	ILIN	Sets illegal instruction bit.
SR[0]	C	Cleared
SR[1]	T	Cleared
SR[5:4]	S[1:0]	Cleared
SR[31]	SLF	Cleared
SR[30:27]	LF[3:0]	Clear loop flags.

Example

illegal

Register/Memory Address	Before	After
SR	\$18E0 0003	\$00E4 0000
EMR	\$0000 0000	\$0000 0001

Instruction Formats and Opcodes

Instruction	Words	Cycles ¹	Type	Opcode
ILLEGAL	1	4	5	<div style="display: flex; justify-content: space-between; align-items: center;"> 15 8 7 0 </div> <div style="border: 1px solid black; padding: 2px; display: flex; justify-content: space-between;"> 1 0 0 1 1 1 1 0 0 1 1 1 1 1 0 0 </div>

Note 1: Cycle count is dependant on the machine state. Typically, five cycles is the service time for an illegal request.

Operation

$D_n \pm (D_a.L * D_b.L) \rightarrow D_n$

Assembler Syntax

IMAC $\pm Da, Db, Dn$

Description

IMAC $\pm Da, Db, Dn$

Performs signed integer-multiplication on the LP contents of two source data registers (Da and Db) and adds or subtracts the product to or from a destination data register (Dn). The default operation is the addition of the product to the destination register.

Status and Conditions that Affect Instruction

None.

Status and Conditions Changed by Instruction

Register Address	Bit Name	Description
Ln	L	Clears the Ln bit in the destination register.
EMR[2]	DOVF	Set if the result cannot be represented in 40 bits.

Example 1

imac d4,d5,d6

Register/Memory Address	Before	After
D4	\$FF FFFF FFFB	
D5	\$00 0000 0003	
L6:D6	\$0:\$00 0000 0008	\$0:\$FF FFFF FFF9
EMR	\$0000 0000	\$0000 0000

```

-5   $FFFB
x 3  $0003
-15  $000F
+8   $0008
-7   $FFF9
    
```

Example 2

imac -d4,d5,d6

Register/Memory Address	Before	After
D4	\$00 1022 002A	
D5	\$FF FF3A 000B	
L6:D6	\$0:\$00 0000 1000	\$0:\$00 0000 0E32
EMR	\$0000 0000	\$0000 0000

```

-42  $002A
x 11  $000B
-462 $FE32
+4096 $1000
3,634 $0E32
    
```

Instruction Formats and Opcodes

Instruction	Words	Cycles	Type	Opcode
IMAC $\pm Da, Db, Dn$	1	1	1	<div style="display: flex; justify-content: space-between; margin: 0;"> 15 8 7 0 </div> <div style="border: 1px solid black; padding: 2px; display: flex; justify-content: space-between;"> 0 * 1 0 1 0 F F F k 0 J J J J J </div>
IMAC $\pm Da, Da, Dn$	1	1	1	<div style="display: flex; justify-content: space-between; margin: 0;"> 15 8 7 0 </div> <div style="border: 1px solid black; padding: 2px; display: flex; justify-content: space-between;"> 0 * 1 0 1 0 F F F 1 1 0 k 1 j j </div>

Note: ** indicates serial grouping encoding.

Instruction Fields

k	Accumulation Notation	
0	add	1 subtract

i,Db

JJJJ
Data Register Pairs

00000	D0,D4	01000	D2,D4	10000	D0,D0	11000	D1,D2
00001	D0,D5	01001	D2,D5	10001	D0,D1	11001	D1,D3
00010	D0,D6	01010	D2,D6	10010	D0,D2	11010	D5,D6
00011	D0,D7	01011	D2,D7	10011	D0,D3	11011	D5,D7
00100	D1,D4	01100	D3,D4	10100	D4,D4	11100	D2,D2
00101	D1,D5	01101	D3,D5	10101	D4,D5	11101	D2,D3
00110	D1,D6	01110	D3,D6	10110	D4,D6	11110	D6,D6
00111	D1,D7	01111	D3,D7	10111	D4,D7	11111	D6,D7

- Notes:**
1. This instruction can specify D8-D15 as operands by using a prefix.
 2. Register pair order can be inverted for clarity because the order of operation is not important for multiply operations.
 3. The JJJJ encoding does not include the pairs: D1-D1, D3-D3, D5-D5, and D7-D7. These are covered in the jj encoding.

Da,Da

jj
Data Register Pairs

00	D1,D1	01	D3,D3	10	D5,D5	11	D7,D7
----	-------	----	-------	----	-------	----	-------

Note: This instruction can specify D8-D15 as operands by using a prefix.

Dn

FFF
Single Source/Destination Data Register

000	D0	010	D2	100	D4	110	D6
001	D1	011	D3	101	D5	111	D7

Note: This instruction can specify D8-D15 as operands by using a prefix.



IMACLHUU Integer Multiply-Accumulate Lower Unsigned By Upper Unsigned (DALU)

Operation

$D_n + (D_a.L * D_b.H) \rightarrow D_n$

Assembler Syntax

IMACLHUU Da, Db, Dn

Description

IMACLHUU Da,Db,Dn

Performs an unsigned integer multiplication of the 16-bit LP of one source data register (Da) with the 16-bit HP of another source data register (Db). It then adds the zero-extended 32-bit product to a destination data register (Dn). This instruction is optimized for multi-precision-multiplication support.

Status and Conditions that Affect Instruction

None.

Status and Conditions Changed by Instruction

Register Address	Bit Name	Description
Ln	L	Clears the Ln bit in the destination register.
EMR[2]	DOVF	Set if the result cannot be represented in 40 bits.

Example

```
imaclhUU d3,d4,d0
```

Register/Memory Address	Before	After
D3	\$00 0000 0002	
D4	\$FF FFC0 0000	
L0:D0	\$0:\$00 0000 0001	\$0:\$00 0001 FF81
EMR		\$0000 0000

```

-65,472$FFC0
x 2    $0002
-----
-130,944$FF80
+1    $0001
-----
-130,943$FF81
    
```



Instruction Formats and Opcodes

Instruction	Words	Cycles	Type	Opcode																																			
IMACLHUU	Da, Db, Dn	2	1	4																																			
<table border="1" style="width: 100%; border-collapse: collapse;"> <tr> <td style="text-align: center;">15</td> <td style="text-align: center;">8</td> <td style="text-align: center;">7</td> <td colspan="3"></td> <td style="text-align: center;">0</td> </tr> <tr> <td style="text-align: center;">0</td> <td style="text-align: center;">0</td> <td style="text-align: center;">1</td> <td style="text-align: center;">1</td> <td style="text-align: center;">1</td> <td style="text-align: center;">0</td> <td style="text-align: center;">0</td> </tr> <tr> <td style="text-align: center;">j</td> <td style="text-align: center;">j</td> <td style="text-align: center;">j</td> <td style="text-align: center;">0</td> <td style="text-align: center;">0</td> <td style="text-align: center;">F</td> <td style="text-align: center;">F</td> </tr> <tr> <td style="text-align: center;">1</td> <td style="text-align: center;">0</td> <td style="text-align: center;">0</td> <td style="text-align: center;">0</td> <td style="text-align: center;">0</td> <td style="text-align: center;">0</td> <td style="text-align: center;">0</td> </tr> <tr> <td style="text-align: center;">0</td> <td style="text-align: center;">0</td> <td style="text-align: center;">0</td> <td style="text-align: center;">0</td> <td style="text-align: center;">0</td> <td style="text-align: center;">J</td> <td style="text-align: center;">J</td> </tr> </table>					15	8	7				0	0	0	1	1	1	0	0	j	j	j	0	0	F	F	1	0	0	0	0	0	0	0	0	0	0	0	J	J
15	8	7				0																																	
0	0	1	1	1	0	0																																	
j	j	j	0	0	F	F																																	
1	0	0	0	0	0	0																																	
0	0	0	0	0	J	J																																	

Instruction Fields

Da **jjj** **Single Source/Destination Data Register**

000	D0	010	D2	100	D4	110	D6
001	D1	011	D3	101	D5	111	D7

Note: This instruction can specify D8-D15 as operands by using a prefix.

Db **JJJ** **Single Source Data Register**

000	D0	010	D2	100	D4	110	D6
001	D1	011	D3	101	D5	111	D7

Note: This instruction can specify D8-D15 as operands by using a prefix.

Dn **FFF** **Single Source/Destination Data Register**

000	D0	010	D2	100	D4	110	D6
001	D1	011	D3	101	D5	111	D7

Note: This instruction can specify D8-D15 as operands by using a prefix.



Operation

$D_n + (D_a.L * D_b.H) \rightarrow D_n$

Assembler Syntax

IMACUS *Da, Db, Dn*

Description

IMACUS *Da, Db, Dn*

Performs a signed integer multiplication of the unsigned 16-bit LP of one source data register (*Da*) with the signed 16-bit HP of another source data register (*Db*). It then adds the sign-extended 32-bit product to a destination data register (*Dn*). This instruction is optimized for multi-precision-multiplication support.

Status and Conditions that Affect Instruction

None.

Status and Conditions Changed by Instruction

Register Address	Bit Name	Description
<i>Ln</i>	L	Clears the <i>Ln</i> bit in the destination register.
EMR[2]	DOVF	Set if the result cannot be represented in 40 bits.

Example

```
imacus d3,d4,d0
```

Register/Memory Address	Before	After
D3	\$00 7CE8 0002	
D4	\$FF FFC0 F0D0	
L0:D0	\$0:\$00 0000 0000	\$0:\$FF FFFF FF80
EMR		\$0000 0000

2	\$0002
x -64	\$FFC0
-128	\$FF80
+0	\$0000
-128	\$FF80

Instruction Formats and Opcodes

Instruction	Words	Cycles	Type	Opcode												
IMACUS Da, Db, Dn	2	1	4	<table border="1" style="margin-left: auto; margin-right: auto;"> <tr> <td style="text-align: center;">15</td> <td style="text-align: center;">8</td> <td style="text-align: center;">7</td> <td style="text-align: center;">0</td> </tr> <tr> <td>0 0 1 1 0 0 0 0</td> <td>j</td> <td>j</td> <td>j 0 0 F F F</td> </tr> <tr> <td>1 0 0 0 0 0 0 0</td> <td>0</td> <td>0 0 0 0 0</td> <td>J J J</td> </tr> </table>	15	8	7	0	0 0 1 1 0 0 0 0	j	j	j 0 0 F F F	1 0 0 0 0 0 0 0	0	0 0 0 0 0	J J J
15	8	7	0													
0 0 1 1 0 0 0 0	j	j	j 0 0 F F F													
1 0 0 0 0 0 0 0	0	0 0 0 0 0	J J J													

Instruction Fields

Da **jjj** **Single Source/Destination Data Register**

000	D0	010	D2	100	D4	110	D6
001	D1	011	D3	101	D5	111	D7

Note: This instruction can specify D8-D15 as operands by using a prefix.

Db **JJJ** **Single Source Data Register**

000	D0	010	D2	100	D4	110	D6
001	D1	011	D3	101	D5	111	D7

Note: This instruction can specify D8-D15 as operands by using a prefix.

Dn **FFF** **Single Source/Destination Data Register**

000	D0	010	D2	100	D4	110	D6
001	D1	011	D3	101	D5	111	D7

Note: This instruction can specify D8-D15 as operands by using a prefix.



Operation

Da.L * Db.L → Dn

Assembler Syntax

IMPY Da, Db, Dn

Description

IMPY Da,Db,Dn

Performs a signed integer multiplication on the low portions of two signed source data registers (Da, Db) and stores the product in a destination data register (Dn).

Status and Conditions that Affect Instruction

None.

Status and Conditions Changed by Instruction

Register Address	Bit Name	Description
Ln	L	Clears the Ln bit in the destination register.

Example

impy d3,d4,d0

Register/Memory Address	Before	After
D3	\$FF FFFF 0202	
D4	\$00 0000 FFFE	
L0:D0		\$0:\$FF FFFF FBFC

```

514   $0202
x  -2  $FFFE
-----
-1028 $FBFC

```

Instruction Formats and Opcodes

Instruction	Words	Cycles	Type	Opcode
IMPY Da, Da, Dn	1	1	1	<div style="display: flex; justify-content: space-between; align-items: center;"> 15 8 7 0 </div> <div style="display: flex; justify-content: space-between; align-items: center;"> 0 * 1 0 1 0 F F F 1 1 1 0 1 j j </div>
IMPY Da, Db, Dn	1	1	1	<div style="display: flex; justify-content: space-between; align-items: center;"> 15 8 7 0 </div> <div style="display: flex; justify-content: space-between; align-items: center;"> 0 * 1 0 1 0 F F F 0 1 J J J J J </div>

Note: ** indicates serial grouping encoding.

struction Fields

Da, Da **jj** **Data Register Pairs**

00	D1,D1	01	D3,D3	10	D5,D5	11	D7,D7
----	-------	----	-------	----	-------	----	-------

Note: This instruction can specify D8-D15 as operands by using a prefix.

Da, Db **JJJJ** **Data Register Pairs**

0000	D0,D4	0100	D2,D4	1000	D0,D0	1100	D1,D2
0001	D0,D5	0101	D2,D5	1001	D0,D1	1101	D1,D3
0010	D0,D6	0110	D2,D6	1010	D0,D2	1110	D5,D6
0011	D0,D7	0111	D2,D7	1011	D0,D3	1111	D5,D7
0100	D1,D4	01100	D3,D4	10100	D4,D4	11100	D2,D2
0101	D1,D5	01101	D3,D5	10101	D4,D5	11101	D2,D3
0110	D1,D6	01110	D3,D6	10110	D4,D6	11110	D6,D6
0111	D1,D7	01111	D3,D7	10111	D4,D7	11111	D6,D7

- Notes:**
1. This instruction can specify D8-D15 as operands by using a prefix.
 2. Register pair order can be inverted for clarity because the order of operation is not important for multiply operations.
 3. The JJJJ encoding does not include the pairs: D1–D1, D3–D3, D5–D5, D7–D7. These are covered in the jj encoding.

Dn **FFF** **Single Source/Destination Data Register**

000	D0	010	D2	100	D4	110	D6
001	D1	011	D3	101	D5	111	D7

Note: This instruction can specify D8-D15 as operands by using a prefix.

IMPY.W Signed Immediate Integer Multiply (DALU) **IMPY.W**

Operation

#s16 * Dn.L → Dn

Assembler Syntax

IMPY.W #s16,Dn $\{-2^{15} \leq s16 < 2^{15}\}$

Description

IMPY.W #s16,Dn

Performs a signed integer multiplication on the low portion of a source data register (Dn) and an immediate signed 16-bit word. It then stores the result in a destination data register (Dn).

Status and Conditions that Affect Instruction

None.

Status and Conditions Changed by Instruction

Register Address	Bit Name	Description
Ln	L	Clears the Ln bit in the destination register.

Example

```
impy.w #$ffe,d3
```

Register/Memory Address	Before	After
immediate	\$FFFE	
d3	\$00 7FFF FFF8	\$00 0000 0010



-8 \$FFF8
x -2 \$FFFE
+16 \$0010

Instruction Formats and Opcodes

Instruction	Words	Cycles	Type	Opcode																																		
IMPY.W #s16,Dn	2	1	4	<div style="display: flex; justify-content: space-between; margin: 0;"> 15 8 7 0 </div> <table border="1" style="width: 100%; border-collapse: collapse; text-align: center;"> <tr> <td style="width: 8px;">0</td><td style="width: 8px;">0</td><td style="width: 8px;">1</td><td style="width: 8px;">1</td><td style="width: 8px;">1</td><td style="width: 8px;">1</td><td style="width: 8px;">1</td><td style="width: 8px;">1</td><td style="width: 8px;">0</td> <td style="width: 8px;">i</td><td style="width: 8px;">i</td><td style="width: 8px;">i</td><td style="width: 8px;">1</td><td style="width: 8px;">0</td><td style="width: 8px;">F</td><td style="width: 8px;">F</td><td style="width: 8px;">F</td> </tr> <tr> <td style="width: 8px;">1</td><td style="width: 8px;">0</td><td style="width: 8px;">0</td><td style="width: 8px;">i</td><td style="width: 8px;">i</td><td style="width: 8px;">i</td><td style="width: 8px;">i</td><td style="width: 8px;">i</td><td style="width: 8px;">i</td> <td style="width: 8px;">i</td><td style="width: 8px;">i</td><td style="width: 8px;">i</td><td style="width: 8px;">i</td><td style="width: 8px;">i</td><td style="width: 8px;">i</td><td style="width: 8px;">i</td><td style="width: 8px;">i</td> </tr> </table>	0	0	1	1	1	1	1	1	0	i	i	i	1	0	F	F	F	1	0	0	i	i	i	i	i	i	i	i	i	i	i	i	i	i
0	0	1	1	1	1	1	1	0	i	i	i	1	0	F	F	F																						
1	0	0	i	i	i	i	i	i	i	i	i	i	i	i	i	i																						

Instruction Fields

Dn	FFF	Single Source/Destination Data Register					
000	D0	010	D2	100	D4	110	D6
001	D1	011	D3	101	D5	111	D7

Note: This instruction can specify D8-D15 as operands by using a prefix.

#s16 iiiiiiiiiiiiii 16-bit signed immediate data



IMPYHLUU

Integer Multiply Upper
Unsigned By Lower Unsigned
(DALU)

IMPYHLUU

Operation

Da.H * Db.L → Dn

Assembler Syntax

IMPYHLUU Da, Db, Dn

Description

IMPYHLUU Da,Db,Dn

Performs an unsigned integer multiplication on the 16-bit HP of one source data register (Da) and the 16-bit LP of another source data register (Db). It then stores the zero-extended 32-bit result in a destination data register (Dn).

Status and Conditions that Affect Instruction

None.

Status and Conditions Changed by Instruction

Register Address	Bit Name	Description
Ln	L	Clears the Ln bit in the destination register.

Example 1

```
impyhluu d4,d3,d0
```

Register/Memory Address	Before	After
D3	\$00 0002 FFFF	
D4	\$FF FFFF FFFE	
L0:D0		\$0:\$00 FFFE 0001

Example 2

```
impyhluu d4,d3,d0
```

Register/Memory Address	Before	After
D3	\$00 0000 FFFF	
D4	\$FF FFFF FFFE	
L0:D0		\$0:\$00 FFFE 0001



Instruction Formats and Opcodes

Instruction	Words	Cycles	Type	Opcode																																
IMPYHLUU Da, Db, Dn	2	1	4	<div style="display: flex; justify-content: space-between; align-items: center;"> 15 8 7 0 </div> <table border="1" style="width: 100%; border-collapse: collapse;"> <tr> <td style="width: 8px;">0</td><td style="width: 8px;">0</td><td style="width: 8px;">1</td><td style="width: 8px;">1</td><td style="width: 8px;">1</td><td style="width: 8px;">0</td><td style="width: 8px;">1</td><td style="width: 8px;">0</td> <td style="width: 8px;">j</td><td style="width: 8px;">j</td><td style="width: 8px;">j</td><td style="width: 8px;">0</td><td style="width: 8px;">0</td><td style="width: 8px;">F</td><td style="width: 8px;">F</td><td style="width: 8px;">F</td> </tr> <tr> <td style="width: 8px;">1</td><td style="width: 8px;">0</td><td style="width: 8px;">0</td><td style="width: 8px;">0</td><td style="width: 8px;">0</td><td style="width: 8px;">0</td><td style="width: 8px;">0</td><td style="width: 8px;">0</td> <td style="width: 8px;">0</td><td style="width: 8px;">0</td><td style="width: 8px;">0</td><td style="width: 8px;">0</td><td style="width: 8px;">0</td><td style="width: 8px;">J</td><td style="width: 8px;">J</td><td style="width: 8px;">J</td> </tr> </table>	0	0	1	1	1	0	1	0	j	j	j	0	0	F	F	F	1	0	0	0	0	0	0	0	0	0	0	0	0	J	J	J
0	0	1	1	1	0	1	0	j	j	j	0	0	F	F	F																					
1	0	0	0	0	0	0	0	0	0	0	0	0	J	J	J																					

Instruction Fields

Da **jjj** **Single Source/Destination Data Register**

000	D0	010	D2	100	D4	110	D6
001	D1	011	D3	101	D5	111	D7

Note: This instruction can specify D8-D15 as operands by using a prefix.

Db **JJJ** **Single Source Data Register**

000	D0	010	D2	100	D4	110	D6
001	D1	011	D3	101	D5	111	D7

Note: This instruction can specify D8-D15 as operands by using a prefix.

Dn **FFF** **Single Source/Destination Data Register**

000	D0	010	D2	100	D4	110	D6
001	D1	011	D3	101	D5	111	D7

Note: This instruction can specify D8-D15 as operands by using a prefix.



Operation

Da.H * Db.L → Dn

Assembler Syntax

IMPYSU Da, Db, Dn

Description

IMPYSU Da,Db,Dn

Performs a signed integer multiplication on the signed 16-bit HP of one source data register (Da) and the unsigned 16-bit LP of a second source data register (Db). It then stores the sign-extended 32-bit result in a destination data register (Dn).

Status and Conditions that Affect Instruction

None.

Status and Conditions Changed by Instruction

Register Address	Bit Name	Description
Ln	L	Clears the Ln bit in the destination register.

Example

```
impysu d3,d5,d1
```

Register/Memory Address	Before	After
D5	\$00 0000 0122	
D3	\$FF FFFF FFFF	
L1:D1		\$0:\$FF FFFF FEDE

Instruction Formats and Opcodes

Instruction	Words	Cycles	Type	Opcode												
IMPYSU Da, Db, Dn	2	1	4	<table border="1" style="margin-left: auto; margin-right: auto;"> <tr> <td style="text-align: center;">15</td> <td style="text-align: center;">8</td> <td style="text-align: center;">7</td> <td style="text-align: center;">0</td> </tr> <tr> <td style="text-align: center;">0 0 1 1 0 1 0 0</td> <td style="text-align: center;">j</td> <td style="text-align: center;">j</td> <td style="text-align: center;">j 0 0 F F F</td> </tr> <tr> <td style="text-align: center;">1 0 0 0 0 0 0 0</td> <td style="text-align: center;">0</td> <td style="text-align: center;">0 0 0 0 0</td> <td style="text-align: center;">J J J</td> </tr> </table>	15	8	7	0	0 0 1 1 0 1 0 0	j	j	j 0 0 F F F	1 0 0 0 0 0 0 0	0	0 0 0 0 0	J J J
15	8	7	0													
0 0 1 1 0 1 0 0	j	j	j 0 0 F F F													
1 0 0 0 0 0 0 0	0	0 0 0 0 0	J J J													

Instruction Fields

Da **jjj** **Single Source/Destination Data Register**

000	D0	010	D2	100	D4	110	D6
001	D1	011	D3	101	D5	111	D7

Note: This instruction can specify D8-D15 as operands by using a prefix.

Db **JJJ** **Single Source Data Register**

000	D0	010	D2	100	D4	110	D6
001	D1	011	D3	101	D5	111	D7

Note: This instruction can specify D8-D15 as operands by using a prefix.

Dn **FFF** **Single Source/Destination Data Register**

000	D0	010	D2	100	D4	110	D6
001	D1	011	D3	101	D5	111	D7

Note: This instruction can specify D8-D15 as operands by using a prefix.



IMPYUU

Integer Multiply Unsigned By Unsigned (DALU)

IMPYUU

Operation

Da.L * Db.L → Dn

Assembler Syntax

IMPYUU Da, Db, Dn

Description

IMPYUU Da,Db,Dn

Performs an unsigned integer multiplication on the 16-bit LP (Da) of one data register and the 16-bit LP of another data register (Db). It then stores the zero-extended 32-bit result in a data register (Dn).

Status and Conditions that Affect Instruction

None.

Status and Conditions Changed by Instruction

Register Address	Bit Name	Description
Ln	L	Clears the Ln bit in the destination register.

Example

impyuu d5,d3,d1

Register/Memory Address	Before	After
D5	\$00 0000 0002	
D3	\$FF FFFF FFFC	
L1:D1		\$0:\$00 0001 FFF8

Instruction Formats and Opcodes

Instruction	Words	Cycles	Type	Opcode												
IMPYUU Da, Db, Dn	2	1	4	<table border="1"> <tr> <td>15</td> <td>8</td> <td>7</td> <td>0</td> </tr> <tr> <td>0 0 1 1 0 1 1 0</td> <td>j j j</td> <td>0 0</td> <td>F F F</td> </tr> <tr> <td>1 0 0 0 0 0 0 0</td> <td>0 0 0 0 0</td> <td>J J J</td> <td></td> </tr> </table>	15	8	7	0	0 0 1 1 0 1 1 0	j j j	0 0	F F F	1 0 0 0 0 0 0 0	0 0 0 0 0	J J J	
15	8	7	0													
0 0 1 1 0 1 1 0	j j j	0 0	F F F													
1 0 0 0 0 0 0 0	0 0 0 0 0	J J J														

Instruction Fields

Da	jjj	Single Source/Destination Data Register					
000	D0	010	D2	100	D4	110	D6
001	D1	011	D3	101	D5	111	D7

Note: This instruction can specify D8-D15 as operands by using a prefix.

JJJ Single Source Data Register

000	D0	010	D2	100	D4	110	D6
001	D1	011	D3	101	D5	111	D7

Note: This instruction can specify D8-D15 as operands by using a prefix.

Dn FFF Single Source/Destination Data Register

000	D0	010	D2	100	D4	110	D6
001	D1	011	D3	101	D5	111	D7

Note: This instruction can specify D8-D15 as operands by using a prefix.



Operation

$D_n + 1 \rightarrow D_n$

Assembler Syntax

INC D_n

Description

INC D_n

Adds one to a data register (D_n).

Note: The assembler maps this instruction to ADD #u5, D_n , where #u5 = 1.

Status and Conditions that Affect Instruction

Register Address	Bit Name	Description
SR[2]	SM	If set, selects 32-bit arithmetic saturation mode.
SR[5:4]	S[1:0]	Scaling mode bits determine which bits in the result are used in the L_n bit calculation.

Status and Conditions Changed by Instruction

Register Address	Bit Name	Description
SR[0]	C	Calculates and updates the carry bit in the status register.
EMR[2]	DOVF	Set if the result cannot be represented in 40 bits, or if the result saturates to 32 bits in arithmetic saturation mode.
L_n	L	If not in arithmetic saturation mode (SR [SM] = 0), calculates and updates the L_n bit in the destination register. If in arithmetic saturation mode (SR [SM] = 1), clears the L_n bit in the destination register.

Example 1

```
inc d0
```

Register/Memory Address	Before	After
L0:D0	\$0:\$FF FFFF FFFF	\$0:\$00 0000 0000
SR	\$00E4 0000	\$00E4 0001
EMR		\$0000 0000



Example 2

```
inc d15
```

Register/Memory Address	Before	After
SR	\$00E0 0004	\$00E0 0004
L15:D15	\$0:\$00 7FFF FFFF	\$0:\$00 7FFF FFFF
EMR		\$0000 0004

Arithmetic saturation mode set, SR[2], 32-bit overflow indicated in EMR[2].

Instruction Formats and Opcodes

Instruction	Words	Cycles	Type	Opcode
INC Dn	1	1	1	<div style="display: flex; justify-content: space-between; align-items: center;"> 15 8 7 0 </div> <div style="border: 1px solid black; padding: 2px; display: flex; justify-content: space-between; align-items: center;"> 0 * 1 1 1 0 F F F 1 0 0 0 0 0 1 </div>

Note: ** indicates serial grouping encoding.

Instruction Fields

Dn	FFF	Single Source/Destination Data Register					
000	D0	010	D2	100	D4	110	D6
001	D1	011	D3	101	D5	111	D7

Note: This instruction can specify D8-D15 as operands by using a prefix.

INC.F Increment HP of a Data Register by One (DALU) INC.F

Operation

$Dn + \$00:00010000 \rightarrow Dn$

Assembler Syntax

`INC.F Dn`

Description

INC.F Dn

Adds one to the HP of a data register (Dn). Can be used to increment a 16-bit fraction.

Status and Conditions that Affect Instruction

Register Address	Bit Name	Description
SR[2]	SM	If set, selects 32-bit arithmetic saturation mode.
SR[5:4]	S[1:0]	Scaling mode bits determine which bits in the result are used in the Ln bit calculation.

Status and Conditions Changed by Instruction

Register Address	Bit Name	Description
SR[0]	C	Calculates and updates the carry bit in the status register.
EMR[2]	DOVF	Set if the result cannot be represented in 40 bits, or if the result saturates to 32 bits in arithmetic saturation mode.
Ln	L	If not in arithmetic saturation mode (SR [SM] = 0), calculates and updates the Ln bit in the destination register. If in arithmetic saturation mode (SR [SM] = 1), clears the Ln bit in the destination register.

Example

```
inc.f d15
```

Register/Memory Address	Before	After
L0:D15	\$0:\$FF FFFF FFFF	\$0:\$00 0000 FFFF
SR	\$00E4 0000	\$00E4 0001
EMR		\$0000 0000

Instruction Formats and Opcodes

Instruction	Words	Cycles	Type	Opcode																
INC.F Dn	1	1	1	<div style="display: flex; justify-content: space-between; align-items: center;"> 15 8 7 0 </div> <table border="1" style="width: 100%; text-align: center;"> <tr> <td>0</td><td>*</td><td>1</td><td>0</td><td>0</td><td>1</td><td>F</td><td>F</td><td>F</td><td>1</td><td>1</td><td>0</td><td>0</td><td>1</td><td>1</td><td>1</td> </tr> </table>	0	*	1	0	0	1	F	F	F	1	1	0	0	1	1	1
0	*	1	0	0	1	F	F	F	1	1	0	0	1	1	1					

Note: ** indicates serial grouping encoding.

Instruction Fields

Dn	FFF	Single Source/Destination Data Register						
	000	D0	010	D2	100	D4	110	D6
	001	D1	011	D3	101	D5	111	D7

Note: This instruction can specify D8-D15 as operands by using a prefix.



Operation

$Rx + 1 \rightarrow Rx$

Assembler Syntax

INCA Rx

Description

INCA Rx

Adds one to an AGU register (Rx). The stack pointer (SP) cannot be used as an operand by this instruction.

Note: The assembler maps this instruction to ADDA #u5,Rx, where #u5 = 1.

Status and Conditions that Affect Instruction

Register Address	Bit Name	Description
MCTL[31:0]	AM3–AM0	Address modification bits when updating R0–R7. Otherwise, the instruction is not affected by MCTL.

Status and Conditions Changed by Instruction

None.

Example 1

```
inca r0
```

Register/Memory Address	Before	After
MCTL	\$0000 0000	
R0	\$074F 312A	\$074F 312B

Example 2

```
inca r0
```

Register/Memory Address	Before	After
MCTL	\$0000 0000	
R0	\$FFFF FFFF	\$0000 0000



INSERT

Insert Bit Field (DALU)

INSERT

Operation

$Db[(width - 1):0] \rightarrow Dn[(offset + width - 1):offset]$

width = #U6; offset = #u6

width = Da[13:8]; offset = Da[5:0]

Assembler Syntax

```
INSERT #U6, #u6, Db, Dn {0 ≤ U6 ≤ 40}
{0 ≤ u6 ≤ 40} [#U6 + #u6 ≤ 40]
```

```
INSERT Da, Db, Dn {0 ≤ Da[5:0] ≤ 40}
{0 ≤ Da[13:8] ≤ 16} {Da[13:8] + Da[5:0]
≤ 40}
```

Description

These operations insert a bit field from a source data register (Db) into the destination data register (Dn). The bits outside of the inserted field in the destination register are unchanged. In addition, the source register is unchanged.

INSERT #U6, #u6, Db, Dn

Uses two immediate unsigned 6-bit integers for the width (#U6) and offset (#u6).

INSERT Da, Db, Dn

Uses a supplemental data register Da for the width (bits 13:8) and the offset (bits 5:0).

Status and Conditions that Affect Instruction

None.

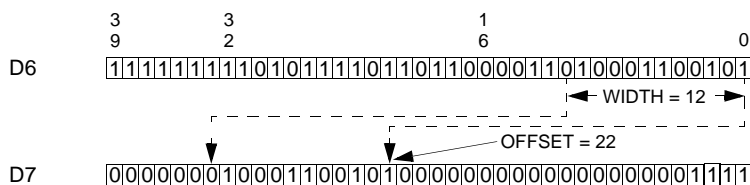
Status and Conditions Changed by Instruction

Register Address	Bit Name	Description
Ln	L	Clears the Ln bit in the destination register.

Example

```
insert #12, #22, d6, d7
```

Register/Memory Address	Before	After
D6	\$FF AF6C 3465	
L7:D7	\$0:\$00 0000 000F	\$0:\$01 1940 000F



Instruction Formats and Opcodes

Instruction	Words	Cycles	Type	Opcode												
INSERT #U6, #u6, Db, Dn	2	1	4	<table border="1"> <tr> <td style="text-align: right;">15</td> <td style="text-align: center;">8</td> <td style="text-align: center;">7</td> <td style="text-align: right;">0</td> </tr> <tr> <td>0 0 1 1 j j j 0</td> <td>1 1 1 0 1 F F F</td> <td></td> <td></td> </tr> <tr> <td>1 0 0 0 l l l l</td> <td>l l i i i i i i</td> <td></td> <td></td> </tr> </table>	15	8	7	0	0 0 1 1 j j j 0	1 1 1 0 1 F F F			1 0 0 0 l l l l	l l i i i i i i		
15	8	7	0													
0 0 1 1 j j j 0	1 1 1 0 1 F F F															
1 0 0 0 l l l l	l l i i i i i i															
INSERT Da, Db, Dn	2	1	4	<table border="1"> <tr> <td style="text-align: right;">15</td> <td style="text-align: center;">8</td> <td style="text-align: center;">7</td> <td style="text-align: right;">0</td> </tr> <tr> <td>0 0 1 1 j j j 0</td> <td>1 0 1 0 1 F F F</td> <td></td> <td></td> </tr> <tr> <td>1 0 0 0 0 0 0 0</td> <td>0 0 0 0 0 J J J</td> <td></td> <td></td> </tr> </table>	15	8	7	0	0 0 1 1 j j j 0	1 0 1 0 1 F F F			1 0 0 0 0 0 0 0	0 0 0 0 0 J J J		
15	8	7	0													
0 0 1 1 j j j 0	1 0 1 0 1 F F F															
1 0 0 0 0 0 0 0	0 0 0 0 0 J J J															

Instruction Fields

Da **JJJ** **Single Source Data Register**

000	D0	010	D2	100	D4	110	D6
001	D1	011	D3	101	D5	111	D7

Note: This instruction can specify D8-D15 as operands by using a prefix.

Db **jjj** **Single Source/Destination Data Register**

000	D0	010	D2	100	D4	110	D6
001	D1	011	D3	101	D5	111	D7

Note: This instruction can specify D8-D15 as operands by using a prefix.

Dn **FFF** **Single Source/Destination Data Register**

000	D0	010	D2	100	D4	110	D6
001	D1	011	D3	101	D5	111	D7

Note: This instruction can specify D8-D15 as operands by using a prefix.

#u6 iiiiii unsigned 6-bit integer

#U6 IIIIII unsigned 6-bit integer

Operation

If $T==0$, then label \rightarrow PC

If $T==0$, then $R_n \rightarrow$ PC

Assembler Syntax

JF label $\{0 \leq \text{label} < 2^{32}, W\}$

JF R_n

Description

If the T bit is cleared, program execution continues at a specified 32-bit memory destination address. If the T bit is set, the PC is updated to point to the next execution set. Program execution continues sequentially. The destination address cannot be in the middle of an execution set.

JF label

Jumps to the absolute memory address specified by a label. The assembler and linker calculate an absolute address from the label.

JF R_n

Jumps to the memory address specified in an address register (R_n). The value of R_n must be word-aligned.

Status and Conditions that Affect Instruction

Register Address	Bit Name	Description
SR[1]	T	True bit

Status and Conditions Changed by Instruction

None.

Example

JF lbl

Instruction	Result
cmpeq.w #35,d1	Not equal, so T bit in SR cleared.
jf lbl move.w #29,d1	Jump to lbl, move.w executed.
inc d1	Skipped over.
move.w #47,d2	Skipped over.
- - - -	Skipped over.
- - - -	Skipped over.
- - - -	Skipped over.
lbl move.w #1A,d4	Execution continues here at lbl.

Register/Memory Address	Before	After
SR	\$00E0 0000	



Register/Memory Address	Before	After
d1	\$00 0000 0000	\$00 0000 0029
d2	\$00 0000 0000	\$00 0000 0000
d4	\$00 0000 0000	\$00 0000 001A
pc	\$0000 0006	\$0000 0016

Instruction Formats and Opcodes

Instruction	Words	Cycles ¹	Type	Opcode																
JF label	3	1/4	3	<table border="1"> <tr> <td style="text-align: center;">15</td> <td style="text-align: center;">8</td> <td style="text-align: center;">7</td> <td style="text-align: center;">0</td> </tr> <tr> <td>0 0 1 1 0 1 1 1</td> <td>A A A</td> <td>a a</td> <td>1 0 0</td> </tr> <tr> <td>0 0 1 A A A A A</td> <td>A A A</td> <td>A A A</td> <td>A A A</td> </tr> <tr> <td>1 0 a a a a a a</td> <td>a a a</td> <td>a a a</td> <td>a a a</td> </tr> </table>	15	8	7	0	0 0 1 1 0 1 1 1	A A A	a a	1 0 0	0 0 1 A A A A A	A A A	A A A	A A A	1 0 a a a a a a	a a a	a a a	a a a
15	8	7	0																	
0 0 1 1 0 1 1 1	A A A	a a	1 0 0																	
0 0 1 A A A A A	A A A	A A A	A A A																	
1 0 a a a a a a	a a a	a a a	a a a																	
JF Rn	1	1/4	4	<table border="1"> <tr> <td style="text-align: center;">15</td> <td style="text-align: center;">8</td> <td style="text-align: center;">7</td> <td style="text-align: center;">0</td> </tr> <tr> <td>1 0 0 1 1 R R R</td> <td>0 1 1</td> <td>0 0 1</td> <td>1 1 1</td> </tr> </table>	15	8	7	0	1 0 0 1 1 R R R	0 1 1	0 0 1	1 1 1								
15	8	7	0																	
1 0 0 1 1 R R R	0 1 1	0 0 1	1 1 1																	

Note 1: If the branch is not taken, it uses 1 cycle. If the branch is taken, it uses 4 cycles.

Instruction Fields

Rn	RRR	Address Register					
000	R0	010	R2	100	R4	110	R6
001	R1	011	R3	101	R5	111	R7

Note: This instruction can specify R8-R15 as operands by using a high register prefix.

label aaaaaaaaaaaaaaaaaAAAAAAAAAAAAAAAAAAAA 32-bit absolute long address

Note: Label must be word-aligned, LSBit = 0.

Operation

If $T==0$, then label \rightarrow PC

If $T==0$, then $R_n \rightarrow$ PC

Assembler Syntax

JFD label [$0 \leq \text{label} < 2^{32}, W$]

JFD R_n

Description

If the T bit is cleared, program execution continues at a specified 32-bit memory destination address after executing the execution set in the delay slot. If the T bit is set, the PC is updated to point to the next execution set and program execution continues sequentially. The destination address cannot be in the middle of an execution set.

JFD label

Jumps to the absolute memory address specified by a label after executing the set in the delay slot. The assembler and linker calculate the destination address from the label.

JFD R_n

Jumps to the memory address specified in an address register (R_n) after executing the execution set in the delay slot.

Status and Conditions that Affect Instruction

Register Address	Bit Name	Description
SR[1]	T	True bit

Status and Conditions changed by Instruction

None.

Example

JFD R0

Instruction	Result
<code>cmpeq.w #35,d1</code>	Not equal, so T bit in SR cleared.
<code>move.w #adr,r0</code>	Places the numerical address of <code>adr</code> in <code>r0</code> .
<code>nop</code>	Delay needed after write to pointer.
<code>jfd r0 move.w #29,d1</code>	Jump to <code>adr</code> , <code>move.w</code> executed.
<code>inc d1</code>	Increment <code>d1</code> to <code>\$2A</code> .
<code>move.w #47,d2</code>	Skipped over.
<code>- - - -</code>	Skipped over.
<code>- - - -</code>	Skipped over.
<code>- - - -</code>	Skipped over.
<code>adr move.w #1A,d4</code>	Execution continues here at <code>lbl</code> .

Register/Memory Address	Before	After
SR	\$00E0 0000	
D1	\$00 0000 0000	\$00 0000 002A
D2	\$00 0000 0000	\$00 0000 0000
D4	\$00 0000 0000	\$00 0000 001A
PC	\$0000 0006	\$0000 0016

Instruction Formats and Opcodes

Instruction	Words	Cycles ¹	Type	Opcode																
JFD label	3	1/4	3	<table border="1"> <tr> <td style="text-align: center;">15</td> <td style="text-align: center;">8</td> <td style="text-align: center;">7</td> <td style="text-align: center;">0</td> </tr> <tr> <td>0 0 1 1 0 1 1 0</td> <td>A A A</td> <td>a a</td> <td>1 0 0</td> </tr> <tr> <td>0 0 1 A A A A A</td> <td>A A A</td> <td>A A A</td> <td>A A A</td> </tr> <tr> <td>1 0 a a a a a a</td> <td>a a a</td> <td>a a a</td> <td>a a a</td> </tr> </table>	15	8	7	0	0 0 1 1 0 1 1 0	A A A	a a	1 0 0	0 0 1 A A A A A	A A A	A A A	A A A	1 0 a a a a a a	a a a	a a a	a a a
15	8	7	0																	
0 0 1 1 0 1 1 0	A A A	a a	1 0 0																	
0 0 1 A A A A A	A A A	A A A	A A A																	
1 0 a a a a a a	a a a	a a a	a a a																	
JFD Rn	1	1/4	4	<table border="1"> <tr> <td style="text-align: center;">15</td> <td style="text-align: center;">8</td> <td style="text-align: center;">7</td> <td style="text-align: center;">0</td> </tr> <tr> <td>1 0 0 1 1 R R R</td> <td>0 1 1</td> <td>0 0 1</td> <td>1 1 0</td> </tr> </table>	15	8	7	0	1 0 0 1 1 R R R	0 1 1	0 0 1	1 1 0								
15	8	7	0																	
1 0 0 1 1 R R R	0 1 1	0 0 1	1 1 0																	

Note 1: If the branch is not taken, it uses 1 cycle. If the branch is taken, it uses 4 cycles minus the time used by the execution set in the delay slot. The cycle count for this instruction cannot be less than 1 cycle.

Instruction Fields

Rn	RRR	Address Register					
	000 R0	010 R2	100 R4	110 R6			
	001 R1	011 R3	101 R5	111 R7			

Note: This instruction can specify R8-R15 as operands by using a high register prefix.

label aaaaaaaaaaaaaaaaaAAAAAAAAAAAAAAAAAAAA 32-bit absolute long address

Note: Label must be word-aligned, LSBit = 0.



JMP

Jump (AGU)

JMP

Operation

label → PC

Rn → PC

Assembler Syntax

JMP label { $0 \leq \text{label} < 2^{32}, W$ }

JMP Rn

Description

These operations continue program execution at a specified 32-bit memory destination address. The destination address cannot be in the middle of an execution set.

JMP label

Jumps to an absolute memory address specified by a label. The assembler and the linker calculate the destination address from the label.

JMP Rn

Jumps to a memory address specified by an address register (Rn). The value in Rn must be word-aligned.

Status and Conditions that Affect Instruction

None.

Status and Conditions Changed by Instruction

None.

Example

```
jmp _label
```

Register/Memory Address	Before	After
_label (absolute)	\$0000 000A	
PC	\$0000 0002	\$0000 000A

Instruction Formats and Opcodes

Instruction	Words	Cycles	Type	Opcode
JMP label	3	3	3	15 8 7 0
				0 0 1 1 0 0 0 1 A A A a a 1 0 0
				0 0 1 A A A A A A A A A A A A A
				1 0 a a a a a a a a a a a a a a
JMP Rn	1	3	4	15 8 7 0
				1 0 0 1 1 R R R 0 1 1 0 0 0 0 1

Instruction Fields

Rn	RRR	Address Register						
	000	R0	010	R2	100	R4	110	R6
	001	R1	011	R3	101	R5	111	R7

Note: This instruction can specify R8-R15 as operands by using a high register prefix.

label aaaaaaaaaaaaaaaaaAAAAAAAAAAAAAAAA 32-bit absolute long address

Note: Label must be word-aligned, LSBit = 0.



JMPD

Jump Using a Delay Slot (AGU)

JMPD

Operation

label → PC

Rn → PC

Assembler Syntax

JMPD label { $0 \leq \text{label} < 2^{32}, W$ }

JMPD Rn

Description

JMPD label

Jumps to an absolute memory destination address specified by a label after executing the execution set in the delay slot. The assembler and the linker calculate the destination address from the label. The destination address cannot be in the middle of an execution set.

JMPD Rn

Jumps to a memory address specified by an address register (Rn) after executing the execution set in the delay slot. The value in Rn must be word-aligned.

Status and Conditions that Affect Instruction

None.

Status and Conditions Changed by Instruction

None.

Example

```
jmpd lbl
```

Instruction	Comment
move.w #\$35,d0	Places \$35 in d0.
jmpd lbl move.w #\$29,d1	Jump to lbl, move.w executed.
inc d1	Increment executed in the delay slot, d1 = \$2A.
move.w #\$20,d3	Skipped over.
- - - -	Skipped over.
- - - -	Skipped over.
- - - -	Skipped over.
lbl move.w #\$16,d4	Execution continues here at lbl.

Instruction Formats and Opcodes

Instruction	Words	Cycles	Type	Opcode
JMPD label	3	3 ¹	3	15 8 7 0
				0 0 1 1 0 0 0 0 A A A a a 1 0 0
				0 0 1 A A A A A A A A A A A A A
				1 0 a a a a a a a a a a a a a a
JMPD Rn	1	3 ¹	4	15 8 7 0
				1 0 0 1 1 R R R 0 1 1 0 0 0 0 0

Note 1: The jump uses 3 cycles minus the execution time used by execution set in the delay slot. The cycle count for this instruction cannot be less than 1 cycle.

Instruction Fields

label aaaaaaaaaaaaaaaaaAAAAAAAAAAAAAAAAAAAA 32-bit absolute long address

Note: Label must be word-aligned, LSBit = 0.

Rn

RRR	Address Register						
000	R0	010	R2	100	R4	110	R6
001	R1	011	R3	101	R5	111	R7

Note: This instruction can specify R8-R15 as operands by using a high register prefix.

Operation

Assembler Syntax

(Next PC) \rightarrow (SP); SR \rightarrow (SP + 4); SP + 8 \rightarrow SP; label \rightarrow PC JSR label { $0 \leq \text{label} < 2^{32}, w$ }

(Next PC) \rightarrow (SP); SR \rightarrow (SP + 4); SP + 8 \rightarrow SP; Rn \rightarrow PC JSR Rn

Description

These operations jump to the subroutine location in program memory that is given by the instruction's effective address. The operation includes an implicit push of the status register (SR) and the program counter (PC) onto the stack. The value of PC stored on the stack is that of the execution set following the current execution set. In addition, the value of the next PC is stored in the RAS shadow register. The destination address cannot be in the middle of an execution set.

JSR label

Jumps to a memory location specified by the label. The assembler and linker calculate the 32-bit absolute destination address from the label.

JSR Rn

Jumps to a memory location contained in an address register (Rn). The value in Rn must be word-aligned.

Status and Conditions that Affect Instruction

Register Address	Bit Name	Description
SR[18]	EXP	Determines which stack pointer is used.

Status and Conditions Changed by Instruction

None.

Example

```
jsr r6
```

Register/Memory Address	Before	After
R6	\$0000 0012	
PC	\$0000 0004	\$0000 0012
SP	\$0000 0100	\$00000108
SR	\$00E0 0000	
(\$00000100)		\$0000 000A
(\$00000104)		\$00E0 0000

JSRD Jump to a Subroutine Using a Delay Slot (AGU) JSRD

Operation

(Next* PC) → (SP); SR → (SP + 4); SP + 8 → SP;
 (Next* PC) → RAS; label → PC

(Next* PC) → (SP); SR → (SP + 4); SP + 8 → SP;
 (Next* PC) → RAS; Rn → PC

Assembler Syntax

JSRD label {0 ≤ label < 2³²,w}

JSRD Rn

Description

Executes the execution set in the delay slot, then pushes the next* PC (the PC of the execution set after the delay slot) and SR onto the stack, and causes program execution to continue at the address defined by label or Rn. In addition, the next* PC is stored in the RAS register, and RAS becomes valid. The destination address cannot be in the middle of an execution set.

JSRD label

Jumps to a memory location specified by an immediate 32-bit absolute address.

JSRD Rn

Jumps to a memory location contained in an address register (Rn). The value in Rn must be word-aligned.

Status and Conditions that Affect Instruction

Register Address	Bit Name	Description
SR[18]	EXP	Determines which stack pointer is used.

Status and Conditions Changed by Instruction

None.

Example

```
jsrd r6
```

Instruction	Comment
move.w #subroutine,r6	Places subroutine label in r6.
move.w #\$35,d0	Places \$35 in d0.
jsrd r6 move.w #\$29,d1	Jump to subroutine, place \$29 in d1.
inc d1	Increment executed in the delay slot, d1=\$2A.
- - - -	Skipped over.
- - - -	Skipped over.
- - - -	Skipped over.
subroutine	Execution continues here at subroutine.
move.w #\$16,d4	

Instruction Formats and Opcodes

Instruction	Words	Cycles	Type	Opcode
JSRD label	3	2/3 ¹	3	15 8 7 0 0 0 1 1 0 0 1 0 A A A a a 1 0 0 0 0 1 A A A A A A A A A A A A A 1 0 a a a a a a a a a a a a a a
				15 8 7 0 1 0 0 1 1 R R R 0 1 1 0 0 0 1 0

Note 1: The jump uses three cycles if the largest cycle time of the instructions grouped with JSRD is 3 or greater. The cycle count of two or three is reduced by the execution time used by the execution set in the delay slot. The cycle count for this instruction cannot be less than one cycle.

Instruction Fields

label aaaaaaaaaaaaaaaaaAAAAAAAAAAAAAAAAAAAA absolute long address

Note: Label must be word-aligned, LSBit = 0.

Rn	RRR	Address Register						
	000	R0	010	R2	100	R4	110	R6
	001	R1	011	R3	101	R5	111	R7

Note: This instruction can specify R8-R15 as operands by using a high register prefix.

Operation

If T=1, then label \rightarrow PC

If T=1, then Rn \rightarrow PC

Assembler Syntax

JT label $\{0 \leq \text{label} < 2^{32}, W\}$

JT Rn

Description

If the T bit is set, these operations continue program execution at a specified 32-bit memory destination address. If the T bit is cleared, the PC is updated to point to the next execution set. Program execution continues sequentially. The destination address cannot be in the middle of an execution set.

JT label

Jumps to the memory location specified by the label. The assembler and linker calculate the 32-bit absolute address from the label.

JT Rn

Jumps to the memory location contained in an address register (Rn). The value in Rn must be word-aligned.

Status and Conditions that Affect Instruction

Register Address	Bit Name	Description
SR[1]	T	True bit

Status and Conditions Changed by Instruction

None.

Example

```
jt r0
```

Register/Memory Address	Before	After
R0	\$0000 0010	
SR	\$00E4 0002	
PC	\$0000 0006	\$0000 0010

Instruction Formats and Opcodes

Instruction	Words	Cycles	Type	Opcode
JT label	3	1/4 ¹	3	15 8 7 0
				0 0 1 1 0 1 0 1 A A A a a 1 0 0
				0 0 1 A A A A A A A A A A A A A
				1 0 a a a a a a a a a a a a a a
JT Rn	1	1/4 ¹	4	15 8 7 0
				1 0 0 1 1 R R R 0 1 1 0 0 1 0 1

Note 1: If not taken, the jump uses 1 cycle. If taken, the jump uses 4 cycles.

Instruction Fields

label aaaaaaaaaaaaaaaaaAAAAAAAAAAAAAAAA absolute long address

Note: Label must be word-aligned, LSBit = 0.

Rn	RRR		Address Register				
	000	R0	010	R2	100	R4	110
001	R1	011	R3	101	R5	111	R7

Note: This instruction can specify R8-R15 as operands by using a high register prefix.

Operation

If T=1, then label → PC

If T=1, then Rn → PC

Assembler Syntax

JTD label {0 ≤ label < 2³², W}

JTD Rn

Description

If the T bit is set, this instruction continues program execution at a specified 32-bit memory destination address after executing the execution set in the delay slot. If the T bit is cleared, the PC is updated to point to the next execution set. Program execution continues sequentially. The destination address cannot be in the middle of an execution set.

JTD label

Jumps to the memory location specified by the label. The assembler and linker calculate the 32-bit absolute address from the label.

JTD Rn

Jumps to the memory location contained in an address register (Rn). The value in Rn must be word-aligned.

Status and Conditions that Affect Instruction

Register Address	Bit Name	Description
SR[1]	T	True bit

Status and Conditions Changed by Instruction

None.

Example

```
jtd r0
```

Instruction	Comment
move.w #ADRES, r0	Load ADRES into r0.
move.w #\$5, d3	Load 5 into d3.
cmpeq.w #\$5, d3	Set the true bit in the status register.
jtd r0 move.w #\$45, d0	Jump to ADRES stored in r0, execute the move.w.
inc d1	Increment executed in the delay slot.
- - - -	Skipped over.
- - - -	Skipped over.
- - - -	Skipped over.
ADDRESS	Execution continues here at ADDRESS.
move.w #\$16, d4	

Instruction Formats and Opcodes

Instruction	Words	Cycles	Type	Opcode																
JTD label	3	1/4 ¹	3	<table border="1"> <tr> <td style="text-align: center;">15</td> <td style="text-align: center;">8</td> <td style="text-align: center;">7</td> <td style="text-align: center;">0</td> </tr> <tr> <td>0 0 1 1 0 1 0 0</td> <td>A A A</td> <td>a a</td> <td>1 0 0</td> </tr> <tr> <td>0 0 1 A A A A A</td> <td>A A A</td> <td>A A A</td> <td>A A A</td> </tr> <tr> <td>1 0 a a a a a a</td> <td>a a a</td> <td>a a a</td> <td>a a a</td> </tr> </table>	15	8	7	0	0 0 1 1 0 1 0 0	A A A	a a	1 0 0	0 0 1 A A A A A	A A A	A A A	A A A	1 0 a a a a a a	a a a	a a a	a a a
15	8	7	0																	
0 0 1 1 0 1 0 0	A A A	a a	1 0 0																	
0 0 1 A A A A A	A A A	A A A	A A A																	
1 0 a a a a a a	a a a	a a a	a a a																	
JTD Rn	1	1/4 ¹	4	<table border="1"> <tr> <td style="text-align: center;">15</td> <td style="text-align: center;">8</td> <td style="text-align: center;">7</td> <td style="text-align: center;">0</td> </tr> <tr> <td>1 0 0 1 1 R R R</td> <td>0 1 1</td> <td>0 0</td> <td>1 0 0</td> </tr> </table>	15	8	7	0	1 0 0 1 1 R R R	0 1 1	0 0	1 0 0								
15	8	7	0																	
1 0 0 1 1 R R R	0 1 1	0 0	1 0 0																	

Note 1: If the jump is not taken, it uses 1 cycle. If the jump is taken, it uses 4 cycles minus the time used by the execution set in the delay slot. The cycle count for this instruction cannot be less than 1 cycle.

Instruction Fields

label aaaaaaaaaaaaaaaaaAAAAAAAAAAAAAAAA absolute long address

Note: Label must be word-aligned, LSBit = 0.

Rn	RRR		Address Register					
	000	R0	010	R2	100	R4	110	R6
	001	R1	011	R3	101	R5	111	R7

Note: This instruction can specify R8-R15 as operands by using a high register prefix.



LPMARKx End-of-Loop Mark (PREFIX) LPMARKx

Operation

If $LC_n > 1$, then $SAn \rightarrow PC$
 $LC_n - 1 \rightarrow LC_n$
 else next $PC \rightarrow PC$
 $0 \rightarrow LC_n$
 $0 \rightarrow LFn$

If $LC_n > 1$, then $SAn \rightarrow PC$
 $LC_n - 1 \rightarrow LC_n$
 else next $PC \rightarrow PC$
 $0 \rightarrow LC_n$
 $0 \rightarrow LFn$

If $LC_n > 1$, then $SAn \rightarrow PC$
 $LC_n - 1 \rightarrow LC_n$
 else next $PC \rightarrow PC$
 $0 \rightarrow LC_n$
 $0 \rightarrow LFn$
 $0 \rightarrow SLF$

If $LC_n > 1$, then $SAn \rightarrow PC$
 $LC_n - 1 \rightarrow LC_n$
 else next $PC \rightarrow PC$
 $0 \rightarrow LC_n$
 $0 \rightarrow LFn$
 $0 \rightarrow SLF$

Disassembler Syntax Only

LPMARKB (long loop)

LPMARKA (external of nested loops)

LPMARKB (short loop of 2 sets)

LPMARKA (short loop of 1 set)

Description

The LPMARK prefix bits are used for hardware loops and perform the operations associated with ending a loop iteration: a conditional jump to the start of the loop (based on the value of LC_n) and a decrement of LC_n . In the case where LC_n indicates the last iteration, these bits disable the active loop and do not jump. The LPMARK bits use the SAn/LC_n register of the active loop as specified by the LF bits in SR.

The LPMARK bits are encoded in the prefix words, and are not independent instructions. They are generated automatically by the assembler at the correct positions based on the LOOPSTART and LOOPEND assembly directives inserted by the programmer. The assembler does not allow the programmer to use LPMARKx.

LPMARKB

For long loops (SLF=0), this prefix bit is placed at LA-2 (two sets before the last set of the loop). It instructs the active loop to decrement LCn and issue a jump delayed operation (with 2 delay slots) to SAn, if LCn is greater than one. If LCn is less than or equal to one, then the LCn register and the LFn bit are cleared. For short loops (SLF=1) of two execution sets, this prefix bit is placed at the first set of the loop (SA).

LPMARKA

For long loops (SLF=0), this prefix bit is placed at LA (the last set of the loop) and is used in special cases. It instructs the active loop machine to decrement LCn and jump to SAn if LCn is greater than one. If LCn is less than or equal to one, then the LCn register and the LFn bit are cleared. This prefix bit is used only in cases where there is a possibility that the loop machine might not be able to identify LPMARKB, which is normally used in long loops. An example is the case of nested loops where the inner loop may be skipped with SKIPLS directly to the LA of the enveloping loop. In case of short loops (SLF=1) of one execution set, LPMARKA is always placed at the first set of the loop (SA).

Table A-17. Combinations of LPMARKx Use

LPMARKA	LPMARKB	LFn	SLF	LCn	Description
no	LPMARKB	set	clear	> 1	LCn decrements by one and a jump with two delay slots to SAn occurs. LPMARKs appearing in the delay slots are ignored.
				≤ 1	LCn and LFn are cleared. The active loop is terminated. Every LPMARKA that appears in the next two delay slots is ignored.
LPMARKA	no	set	clear	> 1	LCn decrements by one and a jump to SAn occurs.
				≤ 1	LCn, LFn, and SLF are cleared. The active loop is terminated.
no	LPMARKB	set	set	> 1	LCn decrements by one and LPMARKs appearing in the next execution set are ignored. A jump with one delay slot to SAn occurs.
				≤ 1	LCn, LFn, and SLF are cleared. The active loop is terminated. LPMARKs appearing in the next execution set are treated.
LPMARKA	no	set	set	> 1	LCn decrements by one and a jump to SAn occurs.
				≤ 1	LCn, LFn, and SLF are cleared. The active loop is terminated.
LPMARKA	LPMARKB	set	clear	> 1	If LPMARKA and LPMARKB appear together, LPMARKA belongs to the inner loop and LPMARKB belongs to the outer loop. If the inner LCn > 1, the LPMARKB is ignored and the LPMARKA is executed. If the inner LCn ≤ 1, the inner LCn and the inner LFn are cleared. The active inner loop is terminated and the LPMARKB is executed.
LPMARKA	LPMARKB	set	set	> 1	

Status and Conditions that Affect LPMARK Execution

The loop flag (LFn), short loop flag (SLFn), and loop counters (LCn) affect the response as described in the description above.

Status and Conditions Changed by LPMARK Execution

The loop flag (LFn) and short loop flag (SLFn) are cleared as described in the operation field.

Example

Insertion of lpmarkb by assembler.

Instruction	Disassembled Instruction	Comments
dosetup0 _lab	dosetup0 *+e	Sets up loop 0 with a start address at _lab.
doen0 d6	doen0 d6	Initializes a long loop with the iteration count from d6.
move.w #1,d1	move.w #<\$1,d1	Puts the number one into d1.
move.w #2,d2	move.w #<\$2,d2	Puts the number two into d2.
move.w #3,d3	move.w #<\$3,d3	Puts the number three into d3.
move.w #4,d4	move.w #<\$4,d4	Puts the number four into d4.
loopstart0		An assembler directive that defines the start of the loop.
_lab inc d1	inc d1	Increments d1 each pass through the loop.
inc d2	lpmarkb inc d2	lpmarkb bit placed in the prefix by the assembler. Increments d2 each pass through the loop.
inc d3	inc d3	Increments d3 each pass through the loop.
inc d4	inc d4	Increments d4 each pass through the loop.
loopend0		An assembler directive that defines the end of the loop.
add d1,d2,d0	add d1,d2,d0	Places the sum of d1 and d2 into d0. If two iterations were selected in d6, d1=3, d2=4, and d0=7.

Prefix Formats and Opcodes

End-of-Loop Mark	Words	Cycles	Type	Opcode												
LPMARKx	1	1	4	<table border="1"> <tr> <td>15</td> <td>8</td> <td>7</td> <td>0</td> </tr> <tr> <td>1 0 0 1 a a a 0</td> <td>1 1 0 p j c c c</td> <td></td> <td></td> </tr> </table>	15	8	7	0	1 0 0 1 a a a 0	1 1 0 p j c c c						
15	8	7	0													
1 0 0 1 a a a 0	1 1 0 p j c c c															
LPMARKx	2	1	4	<table border="1"> <tr> <td>15</td> <td>8</td> <td>7</td> <td>0</td> </tr> <tr> <td>0 0 1 1 a a a 0</td> <td>H t h p j c c c</td> <td></td> <td></td> </tr> <tr> <td>1 0 1 b B e E T</td> <td>b B e E b B e E</td> <td></td> <td></td> </tr> </table>	15	8	7	0	0 0 1 1 a a a 0	H t h p j c c c			1 0 1 b B e E T	b B e E b B e E		
15	8	7	0													
0 0 1 1 a a a 0	H t h p j c c c															
1 0 1 b B e E T	b B e E b B e E															

Note: If LPMARKA is present, j = 1. If LPMARKB is present, p = 1. The other bits shown in the encoding table are independent of LPMARKx.

Operation

If $Da[6:0] > 0$, then $Dn \ll Da[6:0] \rightarrow Dn$
 else $Dn \ggg |Da[6:0]| \rightarrow Dn$

Assembler Syntax

LSLL Da, Dn { $-40 \leq Da[6:0] \leq 40$ }

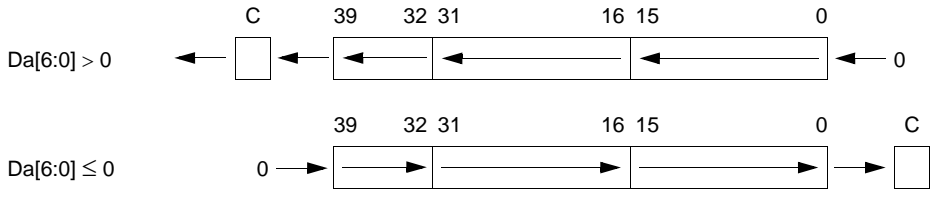
Description

LSLL Da, Dn

Logically shifts a 40-bit data register (Dn) left or right N bits. N is a signed 6-bit integer contained in Da[6:0].

If N is positive, Dn is shifted left. Bit $(40 - N)$ is stored in the C bit. Bits $[(39 - N):0]$ are copied to bits $[39:N]$. Bits $[(N - 1):0]$ are cleared.

If N is negative, Dn is shifted right. Bit $(|N| - 1)$ of Dn is stored in the C bit. Bits $[39:|N|]$ are copied to bits $[(39 - |N|):0]$. Bits $[39:(40 - |N|)]$ are cleared.



Status and Conditions that Affect Instruction

None.

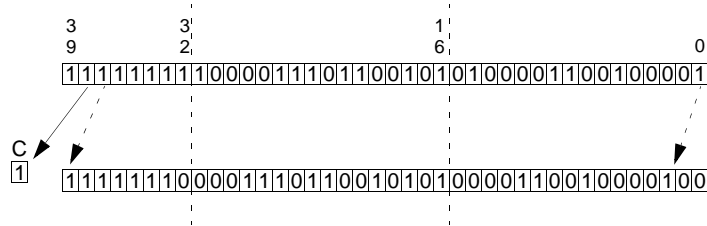
Status and Conditions Changed by Instruction

Register Address	Bit Name	Description
SR[0]	C	Bit $(40 - N)$ of Dn is stored in the C bit for a left shift. Or, bit $(N - 1)$ of Dn is stored in the C bit for a right shift.
Ln	L	Clears the Ln bit in the destination register.

Example 1

```
lsll d4, d2
```

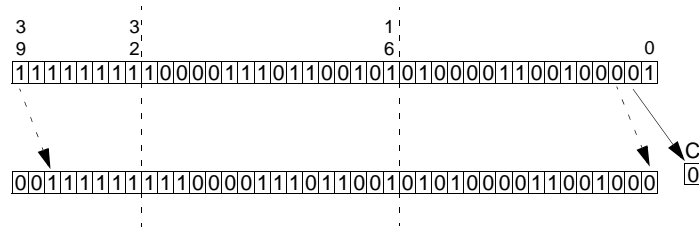
Register/Memory Address	Before	After
D4	\$00 0000 0002	
SR	\$00E4 0000	\$00E4 0001
L2:D2	\$0:\$FF 8765 4321	\$0:\$FE 1D95 0C84



Example 2

```
lsl1 d4,d2
```

Register/Memory Address	Before	After
D4	\$FF FFFF FFFE	
SR	\$00E4 0000	\$00E4 0000
L2:D2	\$0:\$FF 8765 4321	\$0:\$3F E1D9 50C8



Instruction Formats and Opcodes

Instruction	Words	Cycles	Type	Opcode								
LSL Da, Dn	1	1	2	<table border="1" style="display: inline-table;"> <tr> <td style="text-align: center;">15</td> <td style="text-align: center;">8</td> <td style="text-align: center;">7</td> <td style="text-align: center;">0</td> </tr> <tr> <td style="text-align: center;">1 1 0 1 0 1 F F</td> <td style="text-align: center;">F 0 0 0 0 J J J</td> <td></td> <td></td> </tr> </table>	15	8	7	0	1 1 0 1 0 1 F F	F 0 0 0 0 J J J		
15	8	7	0									
1 1 0 1 0 1 F F	F 0 0 0 0 J J J											

Note: ** indicates serial grouping encoding.

Instruction Fields

Da	JJJ	Single Source Data Register					
000	D0	010	D2	100	D4	110	D6
001	D1	011	D3	101	D5	111	D7

Note: This instruction can specify D8-D15 as operands by using a prefix.

Dn	FFF	Single Source/Destination Data Register					
000	D0	010	D2	100	D4	110	D6
001	D1	011	D3	101	D5	111	D7

Note: This instruction can specify D8-D15 as operands by using a prefix.

Operation

$(Dn \ggg 1) \rightarrow Dn; 0 \rightarrow Dn[39]$

Assembler Syntax

LSR Dn

Description

LSR Dn

Shifts the contents of a data register (Dn) right one bit. The LSB (bit 0) is shifted into the carry (C) bit in the status register. Bits [39:1] are copied to bits [38:0]. Bit 39 is cleared.

Status and Conditions that Affect Instruction

None.

Status and Conditions Changed by Instruction

Register Address	Bit Name	Description
SR[0]	C	Dn[0] is stored in the C bit.
Ln	L	Clears the Ln bit in the destination register.

Example

lsr d4

Register/Memory Address	Before	After
SR	\$00E4 0000	\$00E4 0001
L4:D4	\$0:\$FF CCCC CCCD	\$0:\$7F E666 6666

Instruction Formats and Opcodes

Instruction	Words	Cycles	Type	Opcode
LSR Dn	1	1	1	<div style="display: flex; justify-content: space-between; align-items: center;"> 15 8 7 0 </div> <div style="border: 1px solid black; padding: 2px; display: flex; justify-content: space-between; align-items: center;"> 0 * 1 0 0 1 F F F 1 1 0 1 1 1 0 </div>

Note: ** indicates serial grouping encoding.

Instruction Fields

Dn	Single Source/Destination Data Register							
	FFF							
000	D0	010	D2	100	D4	110	D6	
001	D1	011	D3	101	D5	111	D7	

Note: This instruction can specify D8-D15 as operands by using a prefix.



Operation

$(R_x \ggg 1) \rightarrow R_x; 0 \rightarrow R_x[31]$

Assembler Syntax

LSRA Rx

Description

LSRA Rx

Shifts the contents of an AGU register (Rx) right one bit. Bits [31:1] are copied to bits [30:0]. Bit 31 is cleared.

Status and Conditions that Affect Instruction

Register Address	Bit Name	Description
SR[18]	EXP	Determines which stack pointer is used when the stack pointer is an operand. Otherwise, the instruction is not affected by SR.

Status and Conditions Changed by Instruction

None.

Example

```
lsra r2
```

Register/Memory Address	Before	After
R2	\$AAAA AAAA	\$5555 5555

Instruction Formats and Opcodes

Instruction	Words	Cycles	Type	Opcode
LSRA Rx	1	1	2	<div style="display: flex; justify-content: space-between; align-items: center;"> 15 8 7 0 </div> <div style="border: 1px solid black; padding: 2px; display: flex; justify-content: space-between; align-items: center;"> 1 1 1 0 R R R R 1 1 1 1 1 1 1 1 </div>

Instruction Fields

Rx	RRRR	AGU Source/Destination Register						
	0000	N0	0100	—	1000	R0	1100	R4
	0001	N1	0101	—	1001	R1	1101	R5
	0010	N2	0110	—	1010	R2	1110	R6
	0011	N3	0111	SP	1011	R3	1111	R7

Note: This instruction can specify R8-R15 as operands by using a high register prefix.

Operation

If $Da[6:0] > 0$, then $Dn \ggg Da \rightarrow Dn$
 else $Dn \lll |Da| \rightarrow Dn$

$Dn \ggg \#u5 \rightarrow Dn$

Assembler Syntax

`LSRR Da, Dn { -40 ≤ Da[6:0] ≤ 40 }`

`LSRR #u5, Dn { 0 ≤ u5 < 32 }`

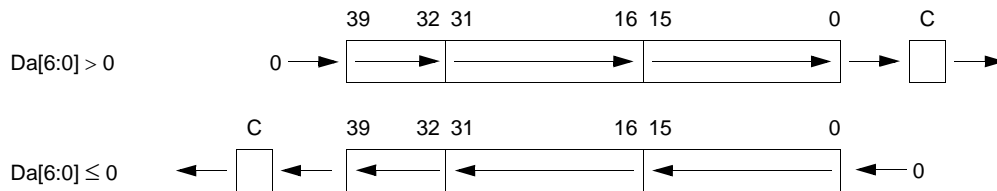
Description

LSRR Da, Dn

Logically shifts the contents of a 40-bit data register (Dn) left or right N bits. N is a signed 6-bit integer contained in Da bits [6:0].

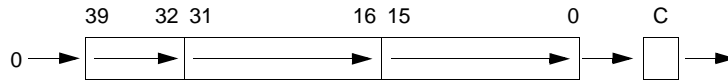
If N is positive, Dn is shifted right. Bit (N – 1) is stored in the C bit. Bits [39:N] are copied to bits [(39 – N):0]. Bits [39:(40 – N)] are cleared.

If N is negative, Dn is shifted left. Bit (40 – |N|) is stored in the C bit. Bits [(39 – |N|):0] are copied to bits [39:|N|]. Bits [(|N| – 1):0] are cleared.



LSRR #u5, Dn

Shifts the contents of a 40-bit data register (Dn) right the number of bits designated in #u5. #u5 is an unsigned 5-bit integer immediate. Bit (N – 1) is stored in the C bit. Bits [39:N] are copied to bits [(39 – N):0]. Bits [39:(40 – N)] are cleared.



Status and Conditions that Affect Instruction

None.

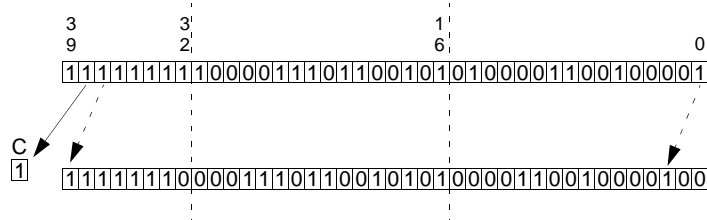
Status and Conditions Changed by Instruction

Register Address	Bit Name	Description
SR[0]	C	Bit (N – 1) of Dn is stored in the C bit for a right shift. Or, bit (40 – N) of Dn is stored in the C bit for a left shift.
Ln	L	Clears the Ln bit in the destination register.

Example 1

lsrr d4,d2

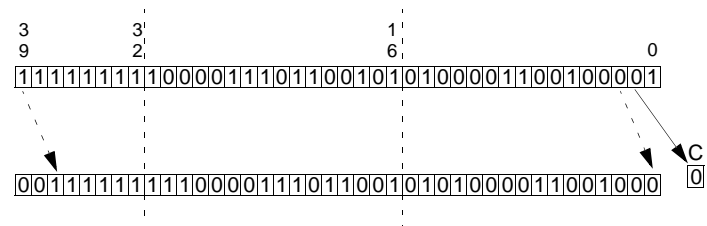
Register/Memory Address	Before	After
D4	\$FF FFFF FFFE	
SR	\$00E4 0000	\$00E4 0001
L2:D2	\$0:\$FF 8765 4321	\$0:\$FE 1D95 0C84



Example 2

lsrr d4,d2

Register/Memory Address	Before	After
D4	\$00 0000 0002	
SR	\$00E4 0000	\$00E4 0000
L2:D2	\$0:\$FF 8765 4321	\$0:\$3F 1ED9 50C8



Instruction Formats and Opcodes

Instruction	Words	Cycles	Type	Opcode																
LSRR Da, Dn	1	1	2	<div style="display: flex; justify-content: space-between; align-items: center;"> 15 8 7 0 </div> <table border="1" style="width: 100%; border-collapse: collapse;"> <tr> <td style="padding: 2px;">1</td> <td style="padding: 2px;">1</td> <td style="padding: 2px;">0</td> <td style="padding: 2px;">1</td> <td style="padding: 2px;">0</td> <td style="padding: 2px;">1</td> <td style="padding: 2px;">F</td> <td style="padding: 2px;">F</td> <td style="padding: 2px;">F</td> <td style="padding: 2px;">0</td> <td style="padding: 2px;">0</td> <td style="padding: 2px;">0</td> <td style="padding: 2px;">1</td> <td style="padding: 2px;">J</td> <td style="padding: 2px;">J</td> <td style="padding: 2px;">J</td> </tr> </table>	1	1	0	1	0	1	F	F	F	0	0	0	1	J	J	J
1	1	0	1	0	1	F	F	F	0	0	0	1	J	J	J					
LSRR #u5, Dn	1	1	2	<div style="display: flex; justify-content: space-between; align-items: center;"> 15 8 7 0 </div> <table border="1" style="width: 100%; border-collapse: collapse;"> <tr> <td style="padding: 2px;">1</td> <td style="padding: 2px;">1</td> <td style="padding: 2px;">0</td> <td style="padding: 2px;">1</td> <td style="padding: 2px;">1</td> <td style="padding: 2px;">1</td> <td style="padding: 2px;">F</td> <td style="padding: 2px;">F</td> <td style="padding: 2px;">F</td> <td style="padding: 2px;">0</td> <td style="padding: 2px;">1</td> <td style="padding: 2px;">i</td> <td style="padding: 2px;">i</td> <td style="padding: 2px;">i</td> <td style="padding: 2px;">i</td> <td style="padding: 2px;">i</td> </tr> </table>	1	1	0	1	1	1	F	F	F	0	1	i	i	i	i	i
1	1	0	1	1	1	F	F	F	0	1	i	i	i	i	i					

Instruction Fields

Da	JJJ	Single Source Data Register					
000	D0	010	D2	100	D4	110	D6
001	D1	011	D3	101	D5	111	D7

Note: This instruction can specify D8-D15 as operands by using a prefix.

Dn	FFF	Single Source/Destination Data Register					
000	D0	010	D2	100	D4	110	D6
001	D1	011	D3	101	D5	111	D7

Note: This instruction can specify D8-D15 as operands by using a prefix.

#u5 iiiii 5-bit unsigned immediate data



LSRW

Word Bitwise Shift Right (DALU)

LSRW

Operation

$Da \ggg 16 \rightarrow Dn$

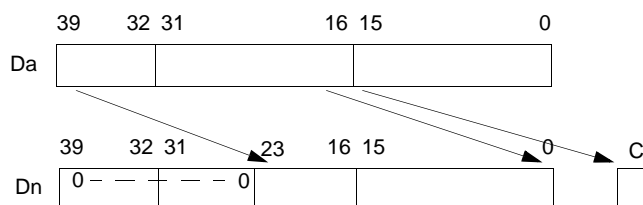
Assembler Syntax

LSRW Da, Dn

Description

LSRW Da, Dn

Copies a source data register (Da) to the destination data register (Dn), logically shifted right 16 bits. Bit 15 of the source register is copied to the C bit. Bits [39:16] of the source register are copied to bits [23:0] of the destination register. Bits [39:24] of the destination register are cleared.



Status and Conditions that Affect Instruction

None.

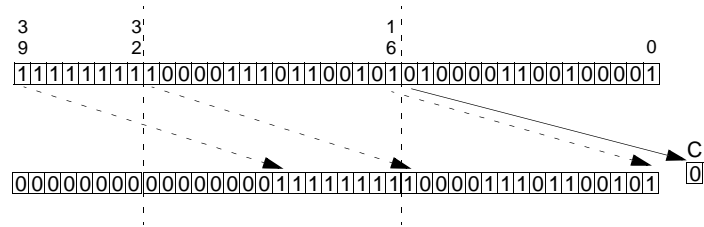
Status and Conditions Changed by Instruction

Register Address	Bit Name	Description
SR[0]	C	Dn[15] is copied into the C bit.
Ln	L	Clears the Ln bit in the destination register.

Example

lsrw d4, d2

Register/Memory Address	Before	After
D4	\$FF 8765 4321	
L2:D2		\$0:\$00 00FF 8765
SR	\$00E4 0000	\$00E4 0000



Instruction Formats and Opcodes

Instruction	Words	Cycles	Type	Opcode																								
LSRW Da, Dn	1	1	2	<table border="1" style="margin-left: auto; margin-right: auto;"> <tr> <td style="text-align: center;">15</td> <td style="text-align: center;">8</td> <td style="text-align: center;">7</td> <td style="text-align: center;">0</td> </tr> <tr> <td style="text-align: center;">1</td> <td style="text-align: center;">1</td> <td style="text-align: center;">0</td> <td style="text-align: center;">1</td> </tr> <tr> <td style="text-align: center;">1</td> <td style="text-align: center;">1</td> <td style="text-align: center;">0</td> <td style="text-align: center;">1</td> </tr> <tr> <td style="text-align: center;">F</td> <td style="text-align: center;">F</td> <td style="text-align: center;">F</td> <td style="text-align: center;">0</td> </tr> <tr> <td style="text-align: center;">0</td> <td style="text-align: center;">0</td> <td style="text-align: center;">0</td> <td style="text-align: center;">1</td> </tr> <tr> <td style="text-align: center;">J</td> <td style="text-align: center;">J</td> <td style="text-align: center;">J</td> <td style="text-align: center;">J</td> </tr> </table>	15	8	7	0	1	1	0	1	1	1	0	1	F	F	F	0	0	0	0	1	J	J	J	J
15	8	7	0																									
1	1	0	1																									
1	1	0	1																									
F	F	F	0																									
0	0	0	1																									
J	J	J	J																									

Instruction Fields

Da	JJJ	Single Source Data Register					
000	D0	010	D2	100	D4	110	D6
001	D1	011	D3	101	D5	111	D7

Note: This instruction can specify D8-D15 as operands by using a prefix.

Dn	FFF	Single Source/Destination Data Register					
000	D0	010	D2	100	D4	110	D6
001	D1	011	D3	101	D5	111	D7

Note: This instruction can specify D8-D15 as operands by using a prefix.



.JAC Signed Fractional Multiply-Accumulate (DALU) MAC

Operation

$Dn + (\#s16 * Da.H) \rightarrow Dn$

$Dn \pm (Da.H * Db.H) \rightarrow Dn$

Assembler Syntax

`MAC #s16, Da, Dn { $-2^{15} \leq s16 < 2^{15}$ }`

`MAC \pm Da, Db, Dn`

Description

These operations perform signed fractional multiplication of two 16-bit signed operands (Da.H and Db.H). They then add or subtract the product to or from a data register (Dn). One operand is the HP of a data register. The other operand is either the HP of a data register or an immediate 16-bit signed data.

MAC #s16, Da, Dn

Adds the product of an immediate 16-bit word and a data register (Da) to the destination register (Dn).

MAC \pm Da, Db, Dn

Multiplies the HP contents of two data registers (Da, Db) and adds or subtracts the product to or from a destination data register (Dn). The default is to add the product to the destination register.

Status and Conditions that Affect Instruction

Register Address	Bit Name	Description
SR[2]	SM	If set, selects 32-bit arithmetic saturation mode.
SR[5:4]	S[1:0]	Scaling mode bits determine which bits in the result are used in the Ln bit calculation.

Status and Conditions Changed by Instruction

Register Address	Bit Name	Description
Ln	L	If not in arithmetic saturation mode (SR [SM] = 0), calculates and updates the Ln bit in the destination register. If in arithmetic saturation mode (SR [SM] = 1), clears the Ln bit in the destination register.
EMR[2]	DOVF	Set if the result cannot be represented in 40 bits, or if the result saturates to 32 bits in arithmetic saturation mode.

Example 1

`mac d4, d5, d6`

Register/Memory Address	Before	After
SR	<code>\$00E0 0000</code>	
D4	<code>\$00 1000 0000</code>	

Register/Memory Address	Before	After
D5	\$00 3000 0000	
L6:D6	\$0:\$00 4000 0000	\$0:\$00 4600 0000
EMR		\$0000 0000

```

0.001 $1000
x 0.011$3000
-----
0.0000110$0600
+0.1000000$4000
-----
0.1000110$4600

```

Example 2

mac # $\$1000$,d5,d6

Register/Memory Address	Before	After
SR	\$00E0 0000	
D5	\$00 3000 261F	
L6:D6	\$0:\$00 4000 0000	\$0:\$00 4600 0000
EMR		\$0000 0000

Instruction Formats and Opcodes

Instruction	Words	Cycles	Type	Opcode																																						
MAC #s16,Da,Dn	2	1	4	<table border="1"> <tr> <td>15</td> <td></td> <td>8</td> <td>7</td> <td></td> <td>0</td> </tr> <tr> <td>0</td> <td>0</td> <td>1</td> <td>1</td> <td>J</td> <td>J</td> <td>J</td> <td>1</td> <td>i</td> <td>i</td> <td>i</td> <td>1</td> <td>0</td> <td>F</td> <td>F</td> <td>F</td> </tr> <tr> <td>1</td> <td>0</td> <td>0</td> <td>i</td> <td>i</td> <td>i</td> <td>i</td> <td>i</td> <td>i</td> <td>i</td> <td>i</td> <td>i</td> <td>i</td> <td>i</td> <td>i</td> <td>i</td> </tr> </table>	15		8	7		0	0	0	1	1	J	J	J	1	i	i	i	1	0	F	F	F	1	0	0	i	i	i	i	i	i	i	i	i	i	i	i	i
15		8	7		0																																					
0	0	1	1	J	J	J	1	i	i	i	1	0	F	F	F																											
1	0	0	i	i	i	i	i	i	i	i	i	i	i	i	i																											
MAC \pm Da,Db,Dn	1	1	1	<table border="1"> <tr> <td>15</td> <td></td> <td>8</td> <td>7</td> <td></td> <td>0</td> </tr> <tr> <td>0</td> <td>*</td> <td>1</td> <td>0</td> <td>0</td> <td>0</td> <td>F</td> <td>F</td> <td>F</td> <td>k</td> <td>0</td> <td>J</td> <td>J</td> <td>J</td> <td>J</td> <td>J</td> </tr> </table>	15		8	7		0	0	*	1	0	0	0	F	F	F	k	0	J	J	J	J	J																
15		8	7		0																																					
0	*	1	0	0	0	F	F	F	k	0	J	J	J	J	J																											
MAC \pm Da,Da,Dn	1	1	1	<table border="1"> <tr> <td>15</td> <td></td> <td>8</td> <td>7</td> <td></td> <td>0</td> </tr> <tr> <td>0</td> <td>*</td> <td>1</td> <td>0</td> <td>1</td> <td>0</td> <td>F</td> <td>F</td> <td>F</td> <td>1</td> <td>1</td> <td>0</td> <td>k</td> <td>0</td> <td>j</td> <td>j</td> </tr> </table>	15		8	7		0	0	*	1	0	1	0	F	F	F	1	1	0	k	0	j	j																
15		8	7		0																																					
0	*	1	0	1	0	F	F	F	1	1	0	k	0	j	j																											

Note: ** indicates serial grouping encoding.

Instruction Fields

k	Accumulation Notation		
0	add	1	subtract

JJJ Single Source Data Register

000	D0	010	D2	100	D4	110	D6
001	D1	011	D3	101	D5	111	D7

Note: This instruction can specify D8-D15 as operands by using a prefix.

Da,Db JJJJ Data Register Pairs

00000	D0,D4	01000	D2,D4	10000	D0,D0	11000	D1,D2
00001	D0,D5	01001	D2,D5	10001	D0,D1	11001	D1,D3
00010	D0,D6	01010	D2,D6	10010	D0,D2	11010	D5,D6
00011	D0,D7	01011	D2,D7	10011	D0,D3	11011	D5,D7
00100	D1,D4	01100	D3,D4	10100	D4,D4	11100	D2,D2
00101	D1,D5	01101	D3,D5	10101	D4,D5	11101	D2,D3
00110	D1,D6	01110	D3,D6	10110	D4,D6	11110	D6,D6
00111	D1,D7	01111	D3,D7	10111	D4,D7	11111	D6,D7

- Notes:**
1. This instruction can specify D8-D15 as operands by using a prefix.
 2. Register pair order can be inverted for clarity because the order of operation is not important for add and multiply operations.
 3. The JJJJ encoding does not include the pairs: D1-D1, D3-D3, D5-D5, and D7-D7. These are covered in the jj encoding.

Da,Da jj Data Register Pairs

00	D1,D1	01	D3,D3	10	D5,D5	11	D7,D7
----	-------	----	-------	----	-------	----	-------

Note: This instruction can specify D8-D15 as operands by using a prefix.

Dn FFF Single Source/Destination Data Register

000	D0	010	D2	100	D4	110	D6
001	D1	011	D3	101	D5	111	D7

Note: This instruction can specify D8-D15 as operands by using a prefix.

#s16 **iiiiiiiiiiiiiiii** 16-bit signed immediate data

Operation

$Rnd(Dn \pm (Da.H * Db.H)) \rightarrow Dn$

Assembler Syntax

MACR $\pm Da, Db, Dn$

Description

MACR $\pm Da, Db, Dn$

This instruction performs signed fractional multiplication of two 16-bit signed operands (Da.H and Db.H). It then adds or subtracts the product to or from a destination data register (Dn) and rounds the final result. Rounding adjusts the LSB of the high part of the destination register according to the value of the low part of the register, and then zeros the low part. The two modes of the round function Rnd (), are described on page A-359.

Status and Conditions that Affect Instruction

Register Address	Bit Name	Description
SR[2]	SM	If set, selects 32-bit arithmetic saturation mode.
SR[3]	RM	Rounding mode
SR[5:4]	S[1:0]	The scaling mode bits determine which bits in the result are used in the Ln bit calculation and which bits are used in rounding.

Status and Conditions Changed by Instruction

Register Address	Bit Name	Description
Ln	L	If not in arithmetic saturation mode (SR [SM] = 0), calculates and updates the Ln bit in the destination register. If in arithmetic saturation mode (SR [SM] = 1), clears the Ln bit in the destination register.
EMR[2]	DOVF	Set if the result cannot be represented in 40 bits, or if the result saturates to 32 bits in arithmetic saturation mode.

Example

```
macr d4,d5,d6
```

Register/Memory Address	Before	After
SR	\$00E0 0000	
D4	\$00 0080 0000	
D5	\$00 0080 0000	
L6:D6	\$0:\$00 0007 0000	\$0:\$00 0008 0000

Register/Memory Address Before

EMR

After

\$0000 0000

```

0.000 0000 1000$0080
x 0.000 0000 1000$0080
0.000 0000 0000 0000 1000$000080000
+0.000 0000 0000 0111 0000$0007
rnd0.000 0000 0000 0111 1000$00078

```

0.000 0000 0000 1000 0000**\$0008**

Instruction Formats and Opcodes

Instruction	Words	Cycles	Type	Opcode								
MACR $\pm Da, Db, Dn$	1	1	1	<table border="1"> <tr> <td>15</td> <td>8</td> <td>7</td> <td>0</td> </tr> <tr> <td>0 * 1 0 0 1 F F</td> <td>F k 0 J J J J J</td> <td></td> <td></td> </tr> </table>	15	8	7	0	0 * 1 0 0 1 F F	F k 0 J J J J J		
15	8	7	0									
0 * 1 0 0 1 F F	F k 0 J J J J J											
MACR $\pm Da, Da, Dn$	1	1	1	<table border="1"> <tr> <td>15</td> <td>8</td> <td>7</td> <td>0</td> </tr> <tr> <td>0 * 1 0 1 0 F F</td> <td>F 1 1 1 1 k j j</td> <td></td> <td></td> </tr> </table>	15	8	7	0	0 * 1 0 1 0 F F	F 1 1 1 1 k j j		
15	8	7	0									
0 * 1 0 1 0 F F	F 1 1 1 1 k j j											

Note: ** indicates serial grouping encoding.

Instruction Fields

k**Accumulation Notation**

0	add	1	subtract
---	-----	---	----------

Da,Db**JJJJ****Data Register Pairs**

00000	D0,D4	01000	D2,D4	10000	D0,D0	11000	D1,D2
00001	D0,D5	01001	D2,D5	10001	D0,D1	11001	D1,D3
00010	D0,D6	01010	D2,D6	10010	D0,D2	11010	D5,D6
00011	D0,D7	01011	D2,D7	10011	D0,D3	11011	D5,D7
00100	D1,D4	01100	D3,D4	10100	D4,D4	11100	D2,D2
00101	D1,D5	01101	D3,D5	10101	D4,D5	11101	D2,D3
00110	D1,D6	01110	D3,D6	10110	D4,D6	11110	D6,D6
00111	D1,D7	01111	D3,D7	10111	D4,D7	11111	D6,D7

- Notes:**
1. This instruction can specify D8-D15 as operands by using a prefix.
 2. Register pair order can be inverted for clarity because the order of operation is not important for multiply operations.
 3. The JJJJ encoding does not include the pairs: D1-D1, D3-D3, D5-D5, and D7-D7. These are covered in the jj encoding.

i, Da

jj
Data Register Pairs

00	D1,D1	01	D3,D3	10	D5,D5	11	D7,D7
----	-------	----	-------	----	-------	----	-------

Note: This instruction can specify D8-D15 as operands by using a prefix.

Dn
FFF
Single Source/Destination Data Register

000	D0	010	D2	100	D4	110	D6
001	D1	011	D3	101	D5	111	D7

Note: This instruction can specify D8-D15 as operands by using a prefix.



Operation

$$Dn + (Dc.H * Dd.L) \rightarrow Dn$$

Assembler Syntax

`MACSU Dc, Dd, Dn`

Description

MACSU Dc,Dd,Dn

Performs signed fractional multiplication of the signed 16-bit HP of one data register (Dc) in a register pair (Dc and Dd) by the unsigned 16-bit LP of the other data register (Dd). It then adds the sign-extended 32-bit product to a destination data register (Dn).

Status and Conditions that Affect Instruction

None.

Status and Conditions Changed by Instruction

Register Address	Bit Name	Description
Ln	L	Clears the Ln bit in the destination register.
EMR[2]	DOVF	Set if the result cannot be represented in 40 bits.

Example

```
macsu d0,d1,d4
```

Register/Memory Address	Before	After
D0	\$FF C000 0000	
D1	\$00 0000 0001	
L4:D4	\$0:\$00 0000 0000	\$0:\$FF FFFF 8000
EMR		\$0000 0000



1.100 \$C000
x 0.000 0000 0000 0001\$0001 (2⁻¹⁵)

1.111 1111 1111 1111 1000

\$FFFF 8000

Instruction Formats and Opcodes

Instruction	Words	Cycles	Type	Opcode																
MACSU Dc, Dd, Dn	1	1	1	<div style="display: flex; justify-content: space-between; align-items: center;"> 15 8 7 0 </div> <table border="1" style="width: 100%; text-align: center;"> <tr> <td style="width: 15%;">0</td> <td style="width: 15%;">*</td> <td style="width: 15%;">1</td> <td style="width: 15%;">0</td> <td style="width: 15%;">0</td> <td style="width: 15%;">0</td> <td style="width: 15%;">F</td> <td style="width: 15%;">F</td> <td style="width: 15%;">F</td> <td style="width: 15%;">1</td> <td style="width: 15%;">1</td> <td style="width: 15%;">1</td> <td style="width: 15%;">0</td> <td style="width: 15%;">0</td> <td style="width: 15%;">e</td> <td style="width: 15%;">e</td> </tr> </table>	0	*	1	0	0	0	F	F	F	1	1	1	0	0	e	e
0	*	1	0	0	0	F	F	F	1	1	1	0	0	e	e					

Note: ** indicates serial grouping encoding.

Instruction Fields

Dc, Dd **ee** **Data Register Pairs**

00	D0, D1	01	D2, D3	10	D4, D5	11	D6, D7
----	--------	----	--------	----	--------	----	--------

Note: This instruction can specify D8-D15 as operands by using a prefix.

Dn **FFF** **Single Source/Destination Data Register**

000	D0	010	D2	100	D4	110	D6
001	D1	011	D3	101	D5	111	D7

Note: This instruction can specify D8-D15 as operands by using a prefix.



.MACUS

Fractional Multiply-Accumulate Unsigned By Signed (DALU)

MACUS

Operation

$D_n + (D_c.L * D_d.H) \rightarrow D_n$

Assembler Syntax

MACUS Dc, Dd, Dn

Description

MACUS Dc,Dd,Dn

Performs signed fractional multiplication of the unsigned 16-bit LP of one data register (Dc) in a register pair by the signed 16-bit HP of the other data register (Dd). It then adds the sign-extended 32-bit product to a data register (Dn).

Status and Conditions that Affect Instruction

None.

Status and Conditions Changed by Instruction

Register Address	Bit Name	Description
Ln	L	Clears the Ln bit in the destination register.
EMR[2]	DOVF	Set if the result cannot be represented in 40 bits.

Example

macus d0,d1,d4

Register/Memory Address	Before	After
D0	\$00 0000 0001	
D1	\$FF C000 0000	
L4:D4	\$0:\$00 3FFF 8000	\$0:\$00 3fff 0000
EMR		\$0000 0000

```

      2-15  $0001
      x 1.100 $C000
      1.111 1111 1111 1111 1000$FFFF 8000
      0.011 1111 1111 1111 1000$3FFF 8000
      -----
0.011 1111 1111 1111 0000$3FFF

```

Instruction Formats and Opcodes

Instruction	Words	Cycles	Type	Opcode																
MACUS Dc, Dd, Dn	1	1	1	<div style="display: flex; justify-content: space-between; align-items: center;"> 15 8 7 0 </div> <table border="1" style="width: 100%; text-align: center;"> <tr> <td>0</td><td>*</td><td>1</td><td>0</td><td>1</td><td>1</td><td>F</td><td>F</td><td>F</td><td>1</td><td>1</td><td>0</td><td>0</td><td>0</td><td>e</td><td>e</td> </tr> </table>	0	*	1	0	1	1	F	F	F	1	1	0	0	0	e	e
0	*	1	0	1	1	F	F	F	1	1	0	0	0	e	e					

Note: ** indicates serial grouping encoding.

Instruction Fields

Dc, Dd	ee	Data Register Pairs							
		00	D0, D1	01	D2, D3	10	D4, D5	11	D6, D7

Note: This instruction can specify D8-D15 as operands by using a prefix.

Dn	FFF	Single Source/Destination Data Register							
		000	D0	010	D2	100	D4	110	D6
		001	D1	011	D3	101	D5	111	D7

Note: This instruction can specify D8-D15 as operands by using a prefix.



MACUU

Fractional Multiply-Accumulate Unsigned By Unsigned (DALU)

MACUU

Operation

$D_n + (D_c.L * D_d.L) \rightarrow D_n$

Assembler Syntax

MACUU Dc, Dd, Dn

Description

MACUU Dc,Dd,Dn

Performs unsigned fractional multiplication of the unsigned 16-bit LP of one data register (Dc) by the unsigned 16-bit LP of the other data register (Dd). It then adds the zero-extended 32-bit product to a data register (Dn).

Status and Conditions that Affect Instruction

None.

Status and Conditions Changed by Instruction

Register Address	Bit Name	Description
Ln	L	Clears the Ln bit in the destination register.
EMR[2]	DOVF	Set if the result cannot be represented in 40 bits.

Example

macuu d2,d3,d1

Register/Memory Address	Before	After
D2	\$00 0000 FFFF	
D3	\$00 0000 FFFF	
L1:D1	\$0:\$00 7FFF FFFF	\$0:\$02 7FFC 0001
EMR		\$0000 0000

```

      1.111 1111 1111 1111$FFFF
      x 1.111 1111 1111 1111$FFFF
1 1.111 1111 1111 1100 0000 0000 0000 0001$01 FFFC 0001
+ 0.111 1111 1111 1111 1111 1111 1111 1111 $00 7FFF FFFF
-----
10 0.111 1111 1111 1100 0000 0000 0000 0001$02 7FFC 0001

```



Instruction Formats and Opcodes

Instruction	Words	Cycles	Type	Opcode																
MACUU Dc, Dd, Dn	1	1	1	<div style="display: flex; justify-content: space-between; align-items: center;"> 15 8 7 0 </div> <table border="1" style="width: 100%; text-align: center;"> <tr> <td>0</td><td>*</td><td>1</td><td>0</td><td>1</td><td>1</td><td>F</td><td>F</td><td>F</td><td>1</td><td>1</td><td>0</td><td>0</td><td>1</td><td>e</td><td>e</td> </tr> </table>	0	*	1	0	1	1	F	F	F	1	1	0	0	1	e	e
0	*	1	0	1	1	F	F	F	1	1	0	0	1	e	e					

Note: ** indicates serial grouping encoding.

Instruction Fields

Dc, Dd **ee** **Data Register Pairs**

00	D0, D1	01	D2, D3	10	D4, D5	11	D6, D7
----	--------	----	--------	----	--------	----	--------

Note: This instruction can specify D8-D15 as operands by using a prefix.

Dn **FFF** **Single Source/Destination Data Register**

000	D0	010	D2	100	D4	110	D6
001	D1	011	D3	101	D5	111	D7

Note: This instruction can specify D8-D15 as operands by using a prefix.



MARK

Push the PC into the Trace Buffer (AGU)

MARK

Operation

PC → trace buffer

Assembler Syntax

MARK

Description

MARK

Writes PC (the address of the MARK instruction) to the trace buffer if the trace buffer is enabled (TMARK bit in the TB_CTRL register is set). It is an EOnCE dedicated instruction used for debugging. This instruction can appear only once in an execution set.

Status and Conditions that Affect Instruction

None.

Status and Conditions Changed by Instruction

None.

Instruction Formats and Opcodes

Instruction	Words	Cycles	Type	Opcode
MARK	1	1	4	<div style="display: flex; justify-content: space-between; align-items: center;"> 15 8 7 0 </div> <div style="border: 1px solid black; padding: 2px; display: flex; justify-content: space-between; align-items: center;"> 1 0 0 1 1 1 1 0 0 1 1 1 0 0 1 0 </div>

Operation

If $Dg > Dh$, then $Dg \rightarrow Dh$

Assembler Syntax

MAX Dg, Dh

Description

MAX Dg,Dh

Writes the larger of two signed values in a data register pair (Dg and Dh) to the second of the two registers (Dh). If the first register is greater than the second, the value of the first register is written to the second. Otherwise, the second register is unchanged. Only certain pairs of registers are allowed; see Instruction Fields below.

Status and Conditions that Affect Instruction

None.

Status and Conditions Changed By Instruction

Register Address	Bit Name	Description
Ln	L	Clears the Ln bit in the destination register.

Example

max d0, d4

Register/Memory Address	Before	After
D0	\$FF FFFF FFF5	
L4:D4	\$0:\$FF FFFF 8000	\$0:\$FF FFFF FFF5

Instruction Formats and Opcodes

Instruction	Words	Cycles	Type	Opcode																								
MAX Dg, Dh	1	1	1	<table border="1" style="display: inline-table; vertical-align: middle;"> <tr> <td style="text-align: center;">15</td> <td style="text-align: center;">8</td> <td style="text-align: center;">7</td> <td style="text-align: center;">0</td> </tr> <tr> <td style="text-align: center;">0</td> <td style="text-align: center;">*</td> <td style="text-align: center;">1</td> <td style="text-align: center;">0</td> </tr> <tr> <td style="text-align: center;">1</td> <td style="text-align: center;">1</td> <td style="text-align: center;">G</td> <td style="text-align: center;">G</td> </tr> <tr> <td style="text-align: center;">0</td> <td style="text-align: center;">1</td> <td style="text-align: center;">1</td> <td style="text-align: center;">1</td> </tr> <tr> <td style="text-align: center;">1</td> <td style="text-align: center;">1</td> <td style="text-align: center;">1</td> <td style="text-align: center;">0</td> </tr> <tr> <td style="text-align: center;">0</td> <td style="text-align: center;">0</td> <td style="text-align: center;">0</td> <td style="text-align: center;">0</td> </tr> </table>	15	8	7	0	0	*	1	0	1	1	G	G	0	1	1	1	1	1	1	0	0	0	0	0
15	8	7	0																									
0	*	1	0																									
1	1	G	G																									
0	1	1	1																									
1	1	1	0																									
0	0	0	0																									

Note: ** indicates serial grouping encoding.

Instruction Fields

Dg,Dh	GG	Data Register Pairs					
00	D0,D4	01	D1,D5	10	D2,D6	11	D3,D7

Note: This instruction can specify D8-D15 as operands by using a prefix.



Operation

If $Dg.H > Dh.H$, then $Dg.H \rightarrow Dh.H$

If $Dg.L > Dh.L$, then $Dg.L \rightarrow Dh.L$

Assembler Syntax

MAX2 Dg, Dh

Description

MAX2 Dg,Dh

Writes the larger of each of the corresponding portions in a data register pair (Dg and Dh) to the second of the two registers (Dh). The high and low portions of the two registers are compared independently as 16-bit signed values and written (or not written) based on the comparison. If the high portion of the first register is greater than the high portion of the second, the value of the high portion of the first register is written to the high portion of the second. Otherwise, the high portion of the second register is unchanged. The same process is applied to the low portions of the two registers with the low portion only being affected. The extension byte is undefined. Only certain pairs of registers are allowed; see Instruction Fields below.

Status and Conditions that Affect Instruction

None.

Status and Conditions Changed by Instruction

Register Address	Bit Name	Description
Ln	L	Clears the Ln bit in the destination register.

Example

```
max2 d0,d4
```

Register/Memory Address	Before	After
D0	\$00 0F43 0023	
L4:D4	\$0:\$00 0FE4 8F22	\$0:\$00 0FE4 0023

Instruction Formats and Opcodes

Instruction	Words	Cycles	Type	Opcode
MAX2 Dg, Dh	1	1	1	<div style="display: flex; justify-content: space-between; align-items: center;"> 15 8 7 0 </div> <div style="border: 1px solid black; padding: 2px; display: flex; justify-content: space-between; align-items: center;"> 0 * 1 0 1 1 G G 0 1 1 1 1 1 1 1 </div>

Note: ** indicates serial grouping encoding.



Instruction Fields

Dg,Dh

GG

Data Register Pairs

00	D0,D4	01	D1,D5	10	D2,D6	11	D3,D7
----	-------	----	-------	----	-------	----	-------

Note: This instruction can specify D8-D15 as operands by using a prefix.



Operation

If $Da.L > Db.L$, then $0 \rightarrow VFn$, $Da.L \rightarrow Db.L$
 else $1 \rightarrow VFn$

Assembler Syntax

MAX2VIT Da, Db

Da	Db	VFn
D4.L	D2.L	VF0
D4.H	D2.H	VF1
D0.L	D6.L	VF2
D0.H	D6.H	VF3
D12.L	D10.L	VF0
D12.H	D10.H	VF1
D8.L	D14.L	VF2
D8.H	D14.H	VF3

Description

These operations independently compare the 16-bit contents of the HP and LP of a data register pair to find the larger value. They copy the larger value to the corresponding portion in the second data register and set or clear Viterbi flags (VF0–VF3 in SR) to indicate which portions are larger. The HP and LP of the two registers are compared separately as 16-bit signed values and the Viterbi flags are set or cleared accordingly. These instructions are similar to MAX2, except they also set Viterbi flags. The MAX2VIT instructions are intended to optimize implementation of the Viterbi decoder algorithm. The MAX2VIT instruction is used with conjunction with the VSL instruction (see page A-423).

MAX2VIT Da,Db

For the low portion comparison, the instruction clears VFn ($n=0, 2$) if the LP of Da is greater than the LP of Db. It then copies the contents of the LP of Da to the LP of Db. It sets VFn ($n=0, 2$) if the LP of Da is not greater than the LP of Db. For the high portion comparison, this instruction clears VFn ($n=1, 3$) if the HP of Da is greater than the HP of Db. It then copies the contents of the HP of Da to the HP of Db. It sets VFn ($n=1, 3$) if the HP of Da is not greater than the HP of Db. The high bank of registers can also be used: D12 and D8 substituted for Da, and D10 and D14 substituted for Db. The encoding for the substitution is done with a prefix.

Status and Conditions that Affect Instruction

None.



Status and Conditions Changed by Instruction

Register Address	Bit Name	Description
SR[8]	VF0	Updated by MAX2VIT D4,D2 and MAX2VIT D12,D10.
SR[9]	VF1	Updated by MAX2VIT D4,D2 and MAX2VIT D12,D10.
SR[10]	VF2	Updated by MAX2VIT D0,D6 and MAX2VIT D8,D14.
SR[11]	VF3	Updated by MAX2VIT D0,D6 and MAX2VIT D8,D14.
Ln	L	Clears the Ln bit in the destination register.

Example

```
max2vit d4,d2
```

Register/Memory Address	Before	After
D4	\$00 0643 1023	
D2	\$00 0564 1F22	\$00 0643 1F22
SR	\$00E4 0000	\$00E4 0100

Instruction Formats and Opcodes

Instruction	Words	Cycles	Type	Opcode																								
MAX2VIT D4,D2	1	1	2	<table border="1"> <tr> <td>15</td> <td>8</td> <td>7</td> <td>0</td> </tr> <tr> <td>1</td> <td>1</td> <td>0</td> <td>1</td> </tr> <tr> <td>0</td> <td>1</td> <td>0</td> <td>0</td> </tr> <tr> <td>0</td> <td>0</td> <td>1</td> <td>0</td> </tr> <tr> <td>0</td> <td>0</td> <td>1</td> <td>1</td> </tr> <tr> <td>0</td> <td>0</td> <td>0</td> <td>0</td> </tr> </table>	15	8	7	0	1	1	0	1	0	1	0	0	0	0	1	0	0	0	1	1	0	0	0	0
15	8	7	0																									
1	1	0	1																									
0	1	0	0																									
0	0	1	0																									
0	0	1	1																									
0	0	0	0																									
MAX2VIT D0,D6	1	1	2	<table border="1"> <tr> <td>15</td> <td>8</td> <td>7</td> <td>0</td> </tr> <tr> <td>1</td> <td>1</td> <td>0</td> <td>1</td> </tr> <tr> <td>0</td> <td>1</td> <td>0</td> <td>0</td> </tr> <tr> <td>0</td> <td>0</td> <td>1</td> <td>0</td> </tr> <tr> <td>0</td> <td>0</td> <td>1</td> <td>1</td> </tr> <tr> <td>0</td> <td>0</td> <td>0</td> <td>1</td> </tr> </table>	15	8	7	0	1	1	0	1	0	1	0	0	0	0	1	0	0	0	1	1	0	0	0	1
15	8	7	0																									
1	1	0	1																									
0	1	0	0																									
0	0	1	0																									
0	0	1	1																									
0	0	0	1																									

Note: This instruction can specify D12, D10, D8, and D14 instead of D4, D2, D0, and D6 by using a prefix.



MAXM Transfer Maximum Absolute Value (DALU) MAXM

Operation

If $|Dg| > |Dh|$, then $Dg \rightarrow Dh$

If $Dg == -Dh$, then $|Dg| \rightarrow Dh$

Assembler Syntax

MAXM Dg, Dh

Description

MAXM Dg,Dh

Compares the absolute values of a data register pair (Dg and Dh). If the absolute value of the first register (Dg) is greater than the absolute value of the second (Dh), the value of the first register is written to the second (Dh). Otherwise, the second register is unchanged. In case Dg and Dh have equal magnitudes but opposite signs, the destination register Dh is written with the positive value.

Status and Conditions that Affect Instruction

None.

Status and Conditions Changed by Instruction

Register Address	Bit Name	Description
Ln	L	Clears the Ln bit in the destination register.

Example

```
maxm d2, d6
```

Register/Memory Address	Before	After
D2	\$FF FFFF FFDD	
L6:D6	\$0:\$00 0000 0022	\$0:\$FF FFFF FFDD

\$FFDD = -35, \$0022 = +34

Instruction Formats and Opcodes

Instruction	Words	Cycles	Type	Opcode
MAXM Dg, Dh	1	1	1	<div style="display: flex; justify-content: space-between; align-items: center;"> 15 8 7 0 </div> <div style="border: 1px solid black; padding: 2px; display: flex; justify-content: space-between; align-items: center;"> 0 * 1 0 1 1 G G 0 1 1 1 1 1 1 0 </div>

Note: ** indicates serial grouping encoding.



Instruction Fields

Dg,Dh	GG	Data Register Pairs						
	00	D0,D4	01	D1,D5	10	D2,D6	11	D3,D7

Note: This instruction can specify D8-D15 as operands by using a prefix.



MIN

Transfer Minimum Signed Value (DALU)

MIN

Operation

If $Dg < Dh$, then $Dg \rightarrow Dh$

Assembler Syntax

MIN Dg, Dh

Description

MIN Dg, Dh

Writes the smaller of two signed values in a data register pair (Dg and Dh) to the second of the two registers (Dh). If the first register is less than the second, the value of the first register is written to the second. Otherwise, the second register is unchanged.

Status and Conditions that Affect Instruction

None.

Status and Conditions Changed by Instruction

Register Address	Bit Name	Description
L_n	L	Clears the L_n bit in the destination register.

Example

```
min d1,d5
```

Register/Memory Address	Before	After
D1	\$00 36AE 3FB4	
L5:D5	\$0:\$00 48FE 4A68	\$0:\$00 36AE 3FB4

Instruction Formats and Opcodes

Instruction	Words	Cycles	Type	Opcode
MIN Dg, Dh	1	1	1	<div style="display: flex; justify-content: space-between; align-items: center;"> 15 8 7 0 </div> <div style="border: 1px solid black; padding: 2px; display: flex; justify-content: space-between; align-items: center;"> *0 * 1 0 1 1 G G 0 1 1 1 1 1 0 1 </div>

Note: ** indicates serial grouping encoding.

Instruction Fields

Dg,Dh	GG	Data Register Pairs					
00	D0,D4	01	D1,D5	10	D2,D6	11	D3,D7

Note: This instruction can specify D8-D15 as operands by using a prefix.

MOVE.2F

Move Two Fractional Words from Memory to a Register Pair (AGU)

MOVE.2F

Operation

(EA) → Da:Db

Assembler Syntax

MOVE.2F (EA), Da:Db { 0 ≤ EA < 2³², L }

Description

MOVE.2F (EA),Da:Db

Moves two signed fractional words from memory to a data register pair (Da:Db). The effective memory address of the two words is contained in an address register with an optional offset or post-increment (EA). Each word is written in the HP of its respective data register, sign-extended, and the LP is zero-filled. The reverse operation (moving from a register pair to memory) is done with saturation. It is described in MOVES.2F.

The first operand (Da) will be moved from the lower memory address (EA). The second operand (Db) will be moved from memory address (EA + 2). In order to maintain this behavior in both big endian and little endian modes, the core will interpret the data bus differently in each mode. See [Section 2.4.1, “SC140 Endian Support,”](#) on page 2-56, for more detail on bus and memory behavior for each mode.

The address register values used with this instruction must be a multiple of 4, long aligned.

	39	32	16	0
Da	SIGN EXTENSION	(EA) OPERAND	ZERO FILL	
Db	SIGN EXTENSION	(EA+2) OPERAND	ZERO FILL	

Status and Conditions that Affect Instruction

Register Address	Bit Name	Description
MCTL[31:0]	AM3–AM0	Address modification bits when updating R0–R7. Otherwise, the instruction is not affected by MCTL.
EMR[16]	BEM	Set if big endian mode, cleared if little endian mode.

Status and Conditions Changed by Instruction

Register Address	Bit Name	Description
Ln	L	Clears the Ln bit in the destination registers.

Example

```
move.2f (r7), d2:d3
```

Register/Memory Address	Before	After
MCTL	\$0000 0000	
R7	\$0000 0050	

Register/Memory Address	Before	After
\$0050	\$6000	
\$0052	\$2000	
L2:D2		\$0:\$00 6000 0000
L3:D3		\$0:\$00 2000 0000

Instruction Formats and Opcodes

Instruction	Words	Cycles	Type	Opcode																
MOVE.2F (EA), Da:Db	1	1 ²	1	<div style="display: flex; justify-content: space-between; align-items: center;"> 15 8 7 0 </div> <table border="1" style="width: 100%; text-align: center;"> <tr> <td style="width: 15%;">0</td> <td style="width: 15%;">*</td> <td style="width: 15%;">0</td> <td style="width: 15%;">1</td> <td style="width: 15%;">1</td> <td style="width: 15%;">h</td> <td style="width: 15%;">h</td> <td style="width: 15%;">1</td> <td style="width: 15%;">0</td> <td style="width: 15%;">1</td> <td style="width: 15%;">M</td> <td style="width: 15%;">M</td> <td style="width: 15%;">M</td> <td style="width: 15%;">R</td> <td style="width: 15%;">R</td> <td style="width: 15%;">R</td> </tr> </table>	0	*	0	1	1	h	h	1	0	1	M	M	M	R	R	R
0	*	0	1	1	h	h	1	0	1	M	M	M	R	R	R					

- Notes:**
- ** indicates serial grouping encoding.
 - When the form (Rn + N0) is used in EA, the cycle count is increased by 1.

Instruction Fields

Da:Db **hh** **Data Register Pairs**

00	D0:D1	01	D2:D3	10	D4:D5	11	D6:D7
----	-------	----	-------	----	-------	----	-------

Note: This instruction can specify D8-D15 as operands by using a prefix. In such a case, all the registers in the group will be high registers.

EA **MMM** **Effective Address Notation**

000	(Rn+N0)	010	(Rn)	100	(Rn)+N0	110	(Rn)+N2
001	(Rn)−	011	(Rn)+	101	(Rn)+N1	111	(Rn)+N3

Rn **RRR** **Address Register**

000	R0	010	R2	100	R4	110	R6
001	R1	011	R3	101	R5	111	R7

Note: This instruction can specify R8-R15 as operands by using a high register prefix.



MOVE.2L

Move Two Integer Longs to/from a Register Pair (AGU)

MOVE.2L

Operation

Da,Db ↔ (EA)

Assembler Syntax

MOVE.2L Da:Db, (EA) { $0 \leq EA < 2^{32}, Q$ }
 MOVE.2L (EA), Da:Db { $0 \leq EA < 2^{32}, Q$ }

Description

These operations move two long words from registers to memory, or from memory to registers.

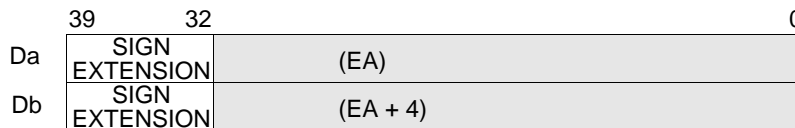
MOVE.2L Da:Db,(EA)

MOVE.2L (EA),Da:Db

Move two long signed integer words from a data register pair (Da:Db) to memory, or from memory to a data register pair. The effective memory address of the two long words is obtained from an address register with an optional offset or post-increment (EA).

The first operand (Da) will be moved to or from the lower memory address (EA). The second operand (Db) will be moved to or from memory address (EA + 4). In order to keep this behavior in both big endian and little endian modes, the core will drive or interpret the data bus differently in each mode. See [Section 2.4.1, “SC140 Endian Support,”](#) on page 2-56, for more detail on bus and memory behavior for each mode.

The address register values used with this instruction must be a multiple of 8, quad word-aligned.



Status and Conditions that Affect Instruction

Register Address	Bit Name	Description
MCTL[31:0]	AM3–AM0	Address modification bits when updating R0–R7. Otherwise, the instruction is not affected by MCTL.
EMR[16]	BEM	Set if big endian mode, cleared if little endian mode.

Status and Conditions Changed by Instruction

Register Address	Bit Name	Description
Ln	L	Clears the Ln bit in the destination registers.

Example

```
move.2l d0:d1, (r0)
```

Register/Memory Address	Before	After
MCTL	\$0000 0000	

Register/Memory Address	Before	After
R0	\$0000 0050	
L0:D0	\$0:\$00 12345 678	
L1:D1	\$0:\$00 5432 9876	
\$0050		\$1234 5678
\$0054		\$5432 9876

Instruction Formats and Opcodes

Instruction	Words	Cycles	Type	Opcode																													
MOVE.2L Da:Db, (EA)	1	1 ¹	2	<table border="1" style="width: 100%; text-align: center;"> <tr> <td style="width: 15%;">15</td> <td style="width: 15%;"></td> <td style="width: 15%;"></td> <td style="width: 15%;"></td> <td style="width: 15%;"></td> <td style="width: 15%;"></td> <td style="width: 15%;"></td> <td style="width: 15%;"></td> <td style="width: 15%;">8</td> <td style="width: 15%;">7</td> <td style="width: 15%;"></td> <td style="width: 15%;"></td> <td style="width: 15%;">0</td> </tr> <tr> <td>1</td> <td>1</td> <td>0</td> <td>0</td> <td>0</td> <td>h</td> <td>h</td> <td>w</td> <td>0</td> <td>0</td> <td>M</td> <td>M</td> <td>M</td> <td>R</td> <td>R</td> <td>R</td> </tr> </table>	15								8	7			0	1	1	0	0	0	h	h	w	0	0	M	M	M	R	R	R
15								8	7			0																					
1	1	0	0	0	h	h	w	0	0	M	M	M	R	R	R																		
MOVE.2L (EA), Da:Db																																	

Note 1: When the form (Rn + N0) is used in EA, the cycle count is increased by 1.

Instruction Fields

w **Read/Write Notation**

0	write	1	read
---	-------	---	------

Da:Db **hh** **Data Register Pairs**

00	D0:D1	01	D2:D3	10	D4:D5	11	D6:D7
----	-------	----	-------	----	-------	----	-------

Note: This instruction can specify D8-D15 as operands by using a prefix. In such a case, all the registers in the group will be high registers.

EA **MMM** **Effective Address Notation**

000	(Rn+N0)	010	(Rn)	100	(Rn)+N0	110	(Rn)+N2
001	(Rn)-	011	(Rn)+	101	(Rn)+N1	111	(Rn)+N3

Rn **RRR** **Address Register**

000	R0	010	R2	100	R4	110	R6
001	R1	011	R3	101	R5	111	R7

Note: This instruction can specify R8-R15 as operands by using a high register prefix.



MOVE.2W

Move Two Integer Words to/from a Register Pair (AGU)

MOVE.2W

Operation

(EA) ↔ Da:Db

Assembler Syntax

MOVE.2W (EA), Da:Db { 0 ≤ EA < 2³², L }
 MOVE.2W Da:Db, (EA) { 0 ≤ EA < 2³², L }

Description

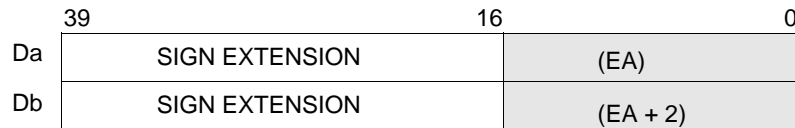
MOVE.2W (EA),Da:Db

MOVE.2W Da:Db,(EA)

Moves two signed integer words from memory to a data register pair (Da:Db), or from the registers to memory. The effective memory address of the two words is obtained from an address register with an optional offset or post-increment (EA). Each word is stored in the LP of its respective data register.

The first operand (Da) will be moved to or from the lower memory address (EA) and the second operand (Db) will be moved to or from memory address (EA + 2). In order to keep this behavior in both big endian and little endian modes, the core will drive or sample the data bus differently in each mode. See [Section 2.4.1, “SC140 Endian Support,”](#) on page 2-56, for more detail on bus and memory behavior for each mode.

The address register values used with this instruction must be a multiple of 4, long word-aligned.



Status and Conditions that Affect Instruction

Register Address	Bit Name	Description
MCTL[31:0]	AM3–AM0	Address modification bits when updating R0–R7. Otherwise, the instruction is not affected by MCTL.
EMR[16]	BEM	Set if big endian mode, cleared if little endian mode.

Status and Conditions Changed by Instruction

Register Address	Bit Name	Description
Ln	L	Clears the Ln bit in the destination registers.

Example

move.2w d0:d1, (r0)

Register/Memory Address	Before	After
MCTL	\$0000 0000	

Register/Memory Address	Before	After
D0	\$FF FFFF AF44	
D1	\$00 0000 2377	
R0	\$0000 0050	
\$0050		\$AF44
\$0052		\$2377

Instruction Formats and Opcodes

Instruction	Words	Cycles	Type	Opcode																												
MOVE.2W (EA), Da:Db	1	1 ²	1	<table border="1" style="width: 100%; text-align: center;"> <tr> <td colspan="4" style="border: none;">15</td> <td colspan="2" style="border: none;">8</td> <td colspan="2" style="border: none;">7</td> <td colspan="4" style="border: none;">0</td> </tr> <tr> <td style="border: none;">0</td> <td style="border: none;">*</td> <td style="border: none;">0</td> <td style="border: none;">w</td> <td style="border: none;">1</td> <td style="border: none;">h</td> <td style="border: none;">h</td> <td style="border: none;">0</td> <td style="border: none;">0</td> <td style="border: none;">1</td> <td style="border: none;">M</td> <td style="border: none;">M</td> <td style="border: none;">M</td> <td style="border: none;">R</td> <td style="border: none;">R</td> <td style="border: none;">R</td> </tr> </table>	15				8		7		0				0	*	0	w	1	h	h	0	0	1	M	M	M	R	R	R
15				8		7		0																								
0	*	0	w	1	h	h	0	0	1	M	M	M	R	R	R																	
MOVE.2W Da:Db, (EA)																																

- Notes:**
- ** indicates serial grouping encoding.
 - When the form (Rn + N0) is used in EA, the cycle count is increased by 1.

Instruction Fields

w

Read/Write Notation			
0	write	1	read

Da:Db

Data Register Pairs							
00	D0:D1	01	D2:D3	10	D4:D5	11	D6:D7

Note: This instruction can specify D8-D15 as operands by using a prefix. In such a case, all the registers in the group will be high registers.

EA

Effective Address Notation							
000	(Rn+N0)	010	(Rn)	100	(Rn)+N0	110	(Rn)+N2
001	(Rn)-	011	(Rn)+	101	(Rn)+N1	111	(Rn)+N3

Rn

Address Register							
000	R0	010	R2	100	R4	110	R6
001	R1	011	R3	101	R5	111	R7

Note: This instruction can specify R8-R15 as operands by using a high register prefix.



MOVE.4F

Move Four Fractional Words from Memory to a Register Quad (AGU)

MOVE.4F

Operation

(EA) → Da:Db:Dc:Dd

Assembler Syntax

MOVE.4F (EA), Da:Db:Dc:Dd {0 ≤ EA < 2³², Q}

Description

MOVE.4F (EA),Da:Db:Dc:Dd

Reads four signed fractional words from memory to a data register quad (Da:Db:Dc:Dd). The effective memory address of the four words is contained in an address register with an optional offset or post-increment (EA). Each word is written into the HP of its respective data register, is sign-extended, and the LP is zero-filled. The reverse operation (moving from a register quad to memory) is done with saturation as described by MOVES.4F.

The first operand (Da) will be moved from the lower memory address (EA). The second operand (Db) will be moved from memory address (EA + 2). The third operand (Dc) will be moved from memory address (EA + 4). And, the fourth operand (Dd) will be moved from memory address (EA + 6). In order to keep this behavior in both big endian and little endian modes, the core will interpret the data bus differently in each mode. See [Section 2.4.1, “SC140 Endian Support,”](#) on page 2-56, for more detail on bus and memory behavior for each mode.

The address register values used with this instruction must be a multiple of 8, quad word-aligned

	39	32	16	0
Da	SIGN EXTENSION		(EA)	ZERO FILL
Db	SIGN EXTENSION		(EA + 2)	ZERO FILL
Dc	SIGN EXTENSION		(EA + 4)	ZERO FILL
Dd	SIGN EXTENSION		(EA + 6)	ZERO FILL

Status and Conditions that Affect Instruction

Register Address	Bit Name	Description
MCTL[31:0]	AM3–AM0	Address modification bits when updating R0–R7. Otherwise, the instruction is not affected by MCTL.
EMR[16]	BEM	Set if big endian mode, cleared if little endian mode.

Status and Conditions Changed by Instruction

Register Address	Bit Name	Description
Ln	L	Clears the Ln bit in the destination registers.

Example

MOVE.4F (r0),d0:d1:d2:d3

Register/Memory Address	Before	After
MCTL	\$0000 0000	
R0	\$0000 0100	
\$0100	\$943C	
\$0102	\$5AB1	
\$0104	\$33E4	
\$0106	\$A7AC	
L0:D0		\$0:\$FF 943C 0000
L1:D1		\$0:\$00 5AB1 0000
L2:D2		\$0:\$00 33E4 0000
L3:D3		\$0:\$FF A7AC 0000

Instruction Formats and Opcodes

Instruction	Words	Cycles	Type	Opcode																																					
MOVE.4F(EA),Da:Db:Dc:Dd	1	1 ²	1	<table border="1" style="width: 100%; text-align: center;"> <tr> <td colspan="15">15</td> <td colspan="2">8</td> <td>7</td> <td colspan="3">0</td> </tr> <tr> <td>0</td><td>*</td><td>0</td><td>0</td><td>1</td><td>k</td><td>0</td><td>1</td> <td>1</td><td>1</td><td>M</td><td>M</td><td>M</td><td>R</td><td>R</td><td>R</td> </tr> </table>	15															8		7	0			0	*	0	0	1	k	0	1	1	1	M	M	M	R	R	R
15															8		7	0																							
0	*	0	0	1	k	0	1	1	1	M	M	M	R	R	R																										

- Notes:**
- ** indicates serial grouping encoding.
 - When the form (Rn + N0) is used in EA, the cycle count is increased by 1.

Instruction Fields

Da:Db:Dc:Dd	k	Data Register Quad			
0	D0:D1:D2:D3	1	D4:D5:D6:D7		

Note: This instruction can specify D8-D15 as operands by using a prefix. In such a case, all the registers in the group will be high registers.

EA	MMM Effective Address Notation							
000	(Rn+N0)	010	(Rn)	100	(Rn)+N0	110	(Rn)+N2	
001	(Rn)-	011	(Rn)+	101	(Rn)+N1	111	(Rn)+N3	

Rn	RRR Address Register							
000	R0	010	R2	100	R4	110	R6	
001	R1	011	R3	101	R5	111	R7	

Note: This instruction can specify R8-R15 as operands by using a high register prefix.



MOVE.4W

Move Four Integer Words to/from a Register Quad (AGU)

MOVE.4W

Operation

(EA) \leftrightarrow Da:Db:Dc:Dd

Assembler Syntax

MOVE.4W (EA), Da:Db:Dc:Dd { $0 \leq EA < 2^{32}, Q$ }
 MOVE.4W Da:Db:Dc:Dd, (EA) { $0 \leq EA < 2^{32}, Q$ }

Description

MOVE.4W (EA),Da:Db:Dc:Dd

MOVE.4W Da:Db:Dc:Dd,(EA)

Moves four signed integer words from memory to a data register quad (Da:Db:Dc:Dd), or from the register quad to memory. The effective memory address of the four words is obtained from an address register with an optional offset or post-increment (EA). Each word is stored in the LP of its respective data register.

The first operand (Da) will be moved to or from the lower memory address (EA). The second operand (Db) will be moved to or from memory address (EA + 2). The third operand (Dc) will be moved to or from memory address (EA + 4). And, the fourth operand (Dd) will be moved to or from memory address (EA + 6). In order to keep this behavior in both big endian and little endian modes, the core will drive or interpret the data bus differently in each mode. See [Section 2.4.1, “SC140 Endian Support,”](#) on page 2-56, for more detail on bus and memory behavior for each mode.

The address register values used with this instruction must be a multiple of 8, quad word-aligned.

	39	16	0
Da	SIGN EXTENSION		(EA)
Db	SIGN EXTENSION		(EA + 2)
Dc	SIGN EXTENSION		(EA + 4)
Dd	SIGN EXTENSION		(EA + 6)

Status and Conditions that Affect Instruction

Register Address	Bit Name	Description
MCTL[31:0]	AM3–AM0	Address modification bits when updating R0–R7. Otherwise, the instruction is not affected by MCTL.
EMR[16]	BEM	Set if big endian mode, cleared if little endian mode.

Status and Conditions Changed by Instruction

Register Address	Bit Name	Description
Ln	L	Clears the Ln bit in the destination registers.

Example

move.4w d0:d1:d2:d3, (r0)

Register/Memory Address	Before	After
MCTL	\$0000 0000	
R0	\$0000 0050	
L0:D0	\$0:\$00 0000 1FEC	
L1:D1	\$0:\$00 0000 2354	
L2:D2	\$0:\$00 0000 38C0	
L3:D3	\$0:\$00 0000 4151	
\$0050		\$1FEC
\$0052		\$2354
\$0054		\$38C0
\$0056		\$4151

Instruction Formats and Opcodes

Instruction	Words	Cycles	Type	Opcode																										
MOVE.4W (EA), Da:Db:Dc:Dd	1	1 ¹	2	<table border="1" style="width: 100%; text-align: center;"> <tr> <td colspan="4" style="border: none;">15</td> <td colspan="2" style="border: none;">8</td> <td colspan="2" style="border: none;">7</td> <td colspan="2" style="border: none;">0</td> </tr> <tr> <td style="border: none;">1</td> <td style="border: none;">1</td> <td style="border: none;">0</td> <td style="border: none;">0</td> <td style="border: none;">1</td> <td style="border: none;">k</td> <td style="border: none;">0</td> <td style="border: none;">w</td> <td style="border: none;">0</td> <td style="border: none;">0</td> <td style="border: none;">M</td> <td style="border: none;">M</td> <td style="border: none;">M</td> <td style="border: none;">R</td> <td style="border: none;">R</td> <td style="border: none;">R</td> </tr> </table>	15				8		7		0		1	1	0	0	1	k	0	w	0	0	M	M	M	R	R	R
15				8		7		0																						
1	1	0	0	1	k	0	w	0	0	M	M	M	R	R	R															
MOVE.4W Da:Db:Dc:Dd, (EA)																														

Note 1: When the form (Rn + N0) is used in EA, the cycle count is increased by 1.

Instruction Fields

w **Read/Write Notation**

0	write	1	read
---	-------	---	------

Da:Db:Dc:Dd **k** **Data Register Quad**

0	D0:D1:D2:D3	1	D4:D5:D6:D7
---	-------------	---	-------------

Note: This instruction can specify D8-D15 as operands by using a prefix. In such a case, all the registers in the group will be high registers.

EA **MMM** **Effective Address Notation**

000	(Rn+N0)	010	(Rn)	100	(Rn)+N0	110	(Rn)+N2
001	(Rn)−	011	(Rn)+	101	(Rn)+N1	111	(Rn)+N3

Rn **RRR** **Address Register**

000	R0	010	R2	100	R4	110	R6
001	R1	011	R3	101	R5	111	R7

Note: This instruction can specify R8-R15 as operands by using a high register prefix.

Operation

(aa) ↔ DR

DR → (aa)

DR → (Rn+s15)

(ea) ↔ DR

(SP+s15) ↔ DR

Assembler Syntax

```
MOVE.B (a16),DR {0 ≤ a16 < 216}
MOVE.B DR,(a16) {0 ≤ a16 < 216}
```

```
MOVE.B DR,(a32) {0 ≤ a32 < 232}
```

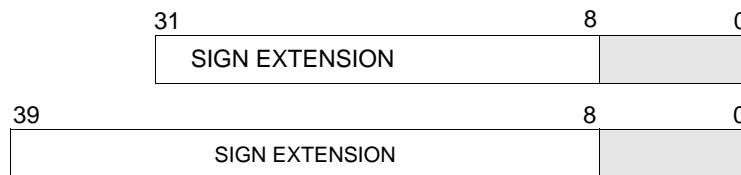
```
MOVE.B DR,(Rn+s15) {-214 ≤ s15 < 214}
```

```
MOVE.B (ea),DR
MOVE.B DR,(ea)
```

```
MOVE.B (SP+s15),DR {-214 ≤ s15 < 214}
MOVE.B DR,(SP+s15) {-214 ≤ s15 < 214}
```

Description

These operations move 8-bit data from memory to a data or address register, or from a register to memory.



MOVE.B (a16),DR

Reads a byte from a 16-bit absolute memory address, sign-extending it into a register.

MOVE.B DR,(a16)

Writes a byte to a 16-bit absolute memory address.

MOVE.B DR,(a32)

Writes a byte to a 32-bit absolute memory address.

MOVE.B DR,(Rn+s15)

Writes a byte to memory from a register. The effective memory address is obtained from an address register with a signed 15-bit offset.

MOVE.B (ea),DR

Reads a byte from memory, sign-extending it into a register. The effective memory address is obtained from an address register with an optional offset or post-increment.

mOVE.B DR,(ea)

Writes a byte to memory. The effective memory address is obtained from an address register with an optional offset or post-increment.

MOVE.B (SP+s15),DR

Reads a byte from memory, sign-extending it into a register. The effective memory address is obtained from the active stack pointer (SP) with a signed 15-bit offset.

MOVE.B DR,(SP+s15)

Writes a byte to memory. The address is obtained from the stack pointer with a signed 15-bit offset.

Status and Conditions that Affect Instruction

Register Address	Bit Name	Description
MCTL[31:0]	AM3–AM0	Address modification bits when updating R0–R7. Otherwise, the instruction is not affected by MCTL.
SR[18]	EXP	Determines which stack pointer is used when the stack pointer is an operand. Otherwise, the instruction is not affected by SR.

Status and Conditions Changed by Instruction

Register Address	Bit Name	Description
Ln	L	Clears the Ln bit in the destination registers.

Example

```
move.b d3, (r7+$3)
```

Register/Memory Address	Before	After
MCTL	\$0000 0000	
D3	\$FF FFFF FFF8	
R7	\$0000 0050	
\$00000053		\$F8



Instruction Formats and Opcodes

Instruction	Words	Cycles	Type	Opcode																
MOVE.B (a16),DR	2	1	3	<table border="1"> <tr> <td>15</td> <td>8</td> <td>7</td> <td>0</td> </tr> <tr> <td>0 0 0 1 H H H H</td> <td>A A A 0 1 1 1 0</td> <td></td> <td></td> </tr> <tr> <td>1 0 0 A A A A A</td> <td>A A A A A A A A</td> <td></td> <td></td> </tr> </table>	15	8	7	0	0 0 0 1 H H H H	A A A 0 1 1 1 0			1 0 0 A A A A A	A A A A A A A A						
15	8	7	0																	
0 0 0 1 H H H H	A A A 0 1 1 1 0																			
1 0 0 A A A A A	A A A A A A A A																			
MOVE.B DR, (a16)	2	1	3	<table border="1"> <tr> <td>15</td> <td>8</td> <td>7</td> <td>0</td> </tr> <tr> <td>0 0 0 0 H H H H</td> <td>A A A 0 1 1 0 0</td> <td></td> <td></td> </tr> <tr> <td>1 0 0 A A A A A</td> <td>A A A A A A A A</td> <td></td> <td></td> </tr> </table>	15	8	7	0	0 0 0 0 H H H H	A A A 0 1 1 0 0			1 0 0 A A A A A	A A A A A A A A						
15	8	7	0																	
0 0 0 0 H H H H	A A A 0 1 1 0 0																			
1 0 0 A A A A A	A A A A A A A A																			
MOVE.B DR, (a32)	3	1	3	<table border="1"> <tr> <td>15</td> <td>8</td> <td>7</td> <td>0</td> </tr> <tr> <td>0 0 0 0 H H H H</td> <td>A A A a a 0 0 1</td> <td></td> <td></td> </tr> <tr> <td>0 0 1 A A A A A</td> <td>A A A A A A A A</td> <td></td> <td></td> </tr> <tr> <td>1 0 a a a a a a</td> <td>a a a a a a a a</td> <td></td> <td></td> </tr> </table>	15	8	7	0	0 0 0 0 H H H H	A A A a a 0 0 1			0 0 1 A A A A A	A A A A A A A A			1 0 a a a a a a	a a a a a a a a		
15	8	7	0																	
0 0 0 0 H H H H	A A A a a 0 0 1																			
0 0 1 A A A A A	A A A A A A A A																			
1 0 a a a a a a	a a a a a a a a																			
MOVE.B DR, (Rn+s15)	2	2	3	<table border="1"> <tr> <td>15</td> <td>8</td> <td>7</td> <td>0</td> </tr> <tr> <td>0 0 0 0 H H H H</td> <td>0 s s 1 0 R R R</td> <td></td> <td></td> </tr> <tr> <td>1 0 0 s s s s s</td> <td>s s s s s s s s</td> <td></td> <td></td> </tr> </table>	15	8	7	0	0 0 0 0 H H H H	0 s s 1 0 R R R			1 0 0 s s s s s	s s s s s s s s						
15	8	7	0																	
0 0 0 0 H H H H	0 s s 1 0 R R R																			
1 0 0 s s s s s	s s s s s s s s																			
MOVE.B (ea),DR	1	1 ¹	4	<table border="1"> <tr> <td>15</td> <td>8</td> <td>7</td> <td>0</td> </tr> <tr> <td>1 0 0 1 H H H H</td> <td>1 1 1 M M R R R</td> <td></td> <td></td> </tr> </table>	15	8	7	0	1 0 0 1 H H H H	1 1 1 M M R R R										
15	8	7	0																	
1 0 0 1 H H H H	1 1 1 M M R R R																			
MOVE.B DR, (ea)	1	1 ¹	4	<table border="1"> <tr> <td>15</td> <td>8</td> <td>7</td> <td>0</td> </tr> <tr> <td>1 0 0 1 H H H H</td> <td>1 0 0 M M R R R</td> <td></td> <td></td> </tr> </table>	15	8	7	0	1 0 0 1 H H H H	1 0 0 M M R R R										
15	8	7	0																	
1 0 0 1 H H H H	1 0 0 M M R R R																			
MOVE.B (SP+s15),DR	2	2	3	<table border="1"> <tr> <td>15</td> <td>8</td> <td>7</td> <td>0</td> </tr> <tr> <td>0 0 0 1 H H H H</td> <td>0 s s 1 1 1 1 0</td> <td></td> <td></td> </tr> <tr> <td>1 0 0 s s s s s</td> <td>s s s s s s s s</td> <td></td> <td></td> </tr> </table>	15	8	7	0	0 0 0 1 H H H H	0 s s 1 1 1 1 0			1 0 0 s s s s s	s s s s s s s s						
15	8	7	0																	
0 0 0 1 H H H H	0 s s 1 1 1 1 0																			
1 0 0 s s s s s	s s s s s s s s																			
MOVE.B DR, (SP+s15)	2	2	3	<table border="1"> <tr> <td>15</td> <td>8</td> <td>7</td> <td>0</td> </tr> <tr> <td>0 0 0 0 H H H H</td> <td>0 s s 1 1 1 0 0</td> <td></td> <td></td> </tr> <tr> <td>1 0 0 s s s s s</td> <td>s s s s s s s s</td> <td></td> <td></td> </tr> </table>	15	8	7	0	0 0 0 0 H H H H	0 s s 1 1 1 0 0			1 0 0 s s s s s	s s s s s s s s						
15	8	7	0																	
0 0 0 0 H H H H	0 s s 1 1 1 0 0																			
1 0 0 s s s s s	s s s s s s s s																			

Note 1: When the form (Rn + N0) is used in ea, the cycle count is increased by 1.

Instruction Fields

DR	HHHH	Data/Address Register					
0000	D0	0100	D4	1000	R0	1100	R4
0001	D1	0101	D5	1001	R1	1101	R5
0010	D2	0110	D6	1010	R2	1110	R6
0011	D3	0111	D7	1011	R3	1111	R7

Note: This instruction can specify D8-D15 or R8-R15 as operands by using a high register prefix.

Operation

#s16 → Db

(aa) → Db

(aa) → Db

(EA) → Db

(Rn+s15) → Db

(SP+s15) → Db

Db → (ea)

Assembler Syntax

MOVE.F #s16,Db { $-2^{15} \leq s16 < 2^{15}$ }

MOVE.F (a16),Db { $0 \leq a16 < 2^{16}, W$ }

MOVE.F (a32),Db { $0 \leq a32 < 2^{32}, W$ }

MOVE.F (EA),Db { $0 \leq EA < 2^{32}, W$ }

MOVE.F (Rn+s15),Db { $-2^{14} \leq s15 < 2^{14}, W$ }

MOVE.F (SP+s15),Db { $-2^{14} \leq s15 < 2^{14}, W$ }

MOVE.F Db, (ea) { $0 \leq ea < 2^{32}, W$ }

Description

These operations read a fractional word from memory into the high portion of a destination data register Db, sign-extended and zero-filled.

This instruction also moves data from a register to memory without saturation. However, moving fractional data from register to memory is generally done with saturation. These instructions are described in MOVES.F.

The address of the access must be word-aligned.



MOVE.F #s16,Db

Loads a 16-bit immediate fractional value into a data register.

MOVE.F (a16),Db

Reads a fractional word from a 16-bit unsigned absolute address into a data register.

MOVE.F (a32),Db

Reads a fractional word from a 32-bit absolute address into a data register.

MOVE.F (EA),Db

Reads a fractional word from memory into a data register. The effective memory address is obtained from an address register with an optional offset or post-increment.

MOVE.F (Rn+s15),Db

Reads a fractional word from memory into a data register. The effective memory address is obtained from an address register with a signed 15-bit offset.

mOVE.F (SP+s15),Db

Reads a fractional word from memory. The effective memory address is obtained from the active stack pointer (SP) with a signed 15-bit offset.

MOVE.F Db,(ea)

Writes an unsaturated fractional word to memory without being affected by the scaling mode. This is the only instruction available for moving the HP of a data register to memory without saturation. The effective memory address is obtained from an address register with an optional offset or post-increment.

Status and Conditions that Affect Instruction

Register Address	Bit Name	Description
MCTL[31:0]	AM3–AM0	Address modification bits when updating R0–R7. Otherwise, the instruction is not affected by MCTL.
SR[18]	EXP	Determines which stack pointer is used when the stack pointer is an operand. Otherwise, the instruction is not affected by SR.

Status and Conditions Changed by Instruction

Register Address	Bit Name	Description
Ln	L	Clears the Ln bit in the destination registers.

Example

```
move.f ($54),d10
```

Register/Memory Address	Before	After
\$00000054	\$6000	
L10:D10		\$0:\$00 6000 0000

Instruction Formats and Opcodes

Instruction	Words	Cycles	Type	Opcode																
MOVE.F #s16,Db	2	1	4	<table border="1"> <tr> <td>15</td> <td>8</td> <td>7</td> <td>0</td> </tr> <tr> <td>0 0 1 0 0 j j j</td> <td>i i i 0 0 0 0 1</td> <td></td> <td></td> </tr> <tr> <td>1 0 0 i i i i i</td> <td>i i i i i i i i</td> <td></td> <td></td> </tr> </table>	15	8	7	0	0 0 1 0 0 j j j	i i i 0 0 0 0 1			1 0 0 i i i i i	i i i i i i i i						
15	8	7	0																	
0 0 1 0 0 j j j	i i i 0 0 0 0 1																			
1 0 0 i i i i i	i i i i i i i i																			
MOVE.F (a16),Db	2	1	3	<table border="1"> <tr> <td>15</td> <td>8</td> <td>7</td> <td>0</td> </tr> <tr> <td>0 0 0 0 0 j j j</td> <td>A A A 0 1 1 0 1</td> <td></td> <td></td> </tr> <tr> <td>1 0 0 A A A A A</td> <td>A A A A A A A A</td> <td></td> <td></td> </tr> </table>	15	8	7	0	0 0 0 0 0 j j j	A A A 0 1 1 0 1			1 0 0 A A A A A	A A A A A A A A						
15	8	7	0																	
0 0 0 0 0 j j j	A A A 0 1 1 0 1																			
1 0 0 A A A A A	A A A A A A A A																			
MOVE.F (a32),Db	3	1	3	<table border="1"> <tr> <td>15</td> <td>8</td> <td>7</td> <td>0</td> </tr> <tr> <td>0 0 0 0 1 j j j</td> <td>A A A a a 0 1 1</td> <td></td> <td></td> </tr> <tr> <td>0 0 1 A A A A A</td> <td>A A A A A A A A</td> <td></td> <td></td> </tr> <tr> <td>1 0 a a a a a a</td> <td>a a a a a a a a</td> <td></td> <td></td> </tr> </table>	15	8	7	0	0 0 0 0 1 j j j	A A A a a 0 1 1			0 0 1 A A A A A	A A A A A A A A			1 0 a a a a a a	a a a a a a a a		
15	8	7	0																	
0 0 0 0 1 j j j	A A A a a 0 1 1																			
0 0 1 A A A A A	A A A A A A A A																			
1 0 a a a a a a	a a a a a a a a																			
MOVE.F (EA),Db	1	1 ²	1	<table border="1"> <tr> <td>15</td> <td>8</td> <td>7</td> <td>0</td> </tr> <tr> <td>0 * 0 1 0 j j j</td> <td>0 1 M M M R R R</td> <td></td> <td></td> </tr> </table>	15	8	7	0	0 * 0 1 0 j j j	0 1 M M M R R R										
15	8	7	0																	
0 * 0 1 0 j j j	0 1 M M M R R R																			
MOVE.F (Rn+s15),Db	2	2	3	<table border="1"> <tr> <td>15</td> <td>8</td> <td>7</td> <td>0</td> </tr> <tr> <td>0 0 0 0 1 j j j</td> <td>1 s s 1 0 R R R</td> <td></td> <td></td> </tr> <tr> <td>1 0 0 s s s s s</td> <td>s s s s s s s s</td> <td></td> <td></td> </tr> </table>	15	8	7	0	0 0 0 0 1 j j j	1 s s 1 0 R R R			1 0 0 s s s s s	s s s s s s s s						
15	8	7	0																	
0 0 0 0 1 j j j	1 s s 1 0 R R R																			
1 0 0 s s s s s	s s s s s s s s																			
MOVE.F (SP+s15),Db	2	2	3	<table border="1"> <tr> <td>15</td> <td>8</td> <td>7</td> <td>0</td> </tr> <tr> <td>0 0 0 0 0 j j j</td> <td>1 s s 1 1 1 0 0</td> <td></td> <td></td> </tr> <tr> <td>1 0 0 s s s s s</td> <td>s s s s s s s s</td> <td></td> <td></td> </tr> </table>	15	8	7	0	0 0 0 0 0 j j j	1 s s 1 1 1 0 0			1 0 0 s s s s s	s s s s s s s s						
15	8	7	0																	
0 0 0 0 0 j j j	1 s s 1 1 1 0 0																			
1 0 0 s s s s s	s s s s s s s s																			
MOVE.F Db,(ea)	1	1 ²	4	<table border="1"> <tr> <td>15</td> <td>8</td> <td>7</td> <td>0</td> </tr> <tr> <td>1 0 0 1 M j j j</td> <td>0 0 1 M 1 R R R</td> <td></td> <td></td> </tr> </table>	15	8	7	0	1 0 0 1 M j j j	0 0 1 M 1 R R R										
15	8	7	0																	
1 0 0 1 M j j j	0 0 1 M 1 R R R																			

- Notes:**
- ** indicates serial grouping encoding.
 - When the form (Rn + N0) is used in EA or ea, the cycle count is increased by 1.

Instruction Fields

Db	jjj	Single Source/Destination Data Register					
000	D0	010	D2	100	D4	110	D6
001	D1	011	D3	101	D5	111	D7

Note: This instruction can specify D8-D15 as operands by using a prefix.

EA	MMM	Effective Address Notation					
000	(Rn+N0)	010	(Rn)	100	(Rn)+N0	110	(Rn)+N2
001	(Rn)−	011	(Rn)+	101	(Rn)+N1	111	(Rn)+N3

Operation

#s32 → C4

#u32 → C1

C4 ↔ Db

C2 ↔ Db

Assembler Syntax

MOVE.L #s32,C4 { $-2^{31} \leq s32 < 2^{31}$ }

MOVE.L #u32,C1 { $0 \leq u32 < 2^{32}$ }

MOVE.L C4,Db
MOVE.L Db,C4

MOVE.L C2,Db
MOVE.L Db,C2

Description

These operations move an immediate long word (32-bit data) into a register, or move a long word between registers. MOVE.L instructions that write to a data register clear the destination register's limit tag bit (Ln bit).



MOVE.L #s32,C4

Loads an immediate signed long word into a general register.

MOVE.L #u32,C1

Loads an immediate unsigned long word into a control register.

MOVE.L C4,Db MOVE.L Db,C4

Moves a long word between a selected data register and a selected general register.

MOVE.L C2,Db MOVE.L Db,C2

Moves a long word between a selected data register and a selected general register.

Status and Conditions Changed by Instruction

Register Address	Bit Name	Description
Ln	L	Clears the Ln bit in the destination register.

Instruction Formats and Opcodes

Instruction	Words	Cycles	Type	Opcode
MOVE.L #s32,C4	3	1	3	<div style="display: flex; justify-content: space-between; align-items: center;"> 15 8 7 0 </div> <div style="border: 1px solid black; padding: 2px; display: flex; justify-content: space-between; align-items: center;"> 0 0 1 1 D D D D i i i l l 0 D 0 </div>

0	0	1	i	i	i	i	i	i	i	i	i	i	i	i	i
1	0														

MOVE.L #u32,C1	3	1	3	15	8	7	0												
				0	0	1	1	0	C	C	C	i	i	i			0	1	1
				0	0	1	i	i	i	i	i	i	i	i	i	i	i	i	i
				1	0														

MOVE.L C4,Db	1	1	2	15	8	7	0												
MOVE.L Db,C4				1	1	0	0	D	D	D	D	0	1	0	D	w	j	j	j

MOVE.L C2,Db	1	1	2	15	8	7	0												
MOVE.L Db,C2				1	1	0	0	C	C	C	C	0	1	1	0	w	j	j	j

Instruction Fields

C1 CCC Control Registers

000	EMR	010	-	100	—	110	—
001	VBA	011	-	101	SR	111	MCTL

C2 CCCC General Registers

0000	EMR	0100	—	1000	SA0	1100	SA2
0001	VBA	0101	SR	1001	LC0	1101	LC2
0010	-	0110	—	1010	SA1	1110	SA3
0011	-	0111	MCTL	1011	LC1	1111	LC3

C4 DDDDD General Registers

00000	D0	01000	D4	10000	R0	11000	R4
00001	B0	01001	B4	10001	N0	11001	M0
00010	D1	01010	D5	10010	R1	11010	R5
00011	B1	01011	B5	10011	N1	11011	M1
00100	D2	01100	D6	10100	R2	11100	R6
00101	B2	01101	B6	10101	N2	11101	M2
00110	D3	01110	D7	10110	R3	11110	R7
00111	B3	01111	B7	10111	N3	11111	M3

Note: If registers D8–D15 or R8–R15 are accessed instead of D0–D7 or R0–R7, a prefix is used.

Db jjj Single Source/Destination Data Register

000	D0	010	D2	100	D4	110	D6
001	D1	011	D3	101	D5	111	D7

Note: If registers D8–D15 are accessed instead of D0–D7, a prefix is used.

MOVE.L Move Long Register Extensions (AGU) MOVE.L

Operation

$((SP+s15)[8:0]) \rightarrow De.E$

$Da.E:Db.E \rightarrow (SP+s15)$

$((SP+s15)[24:16]) \rightarrow Do.E$

$(aa[8:0]) \rightarrow De.E$

$Da.E:Db.E \rightarrow (aa)$

$(aa[24:16]) \rightarrow Do.E$

Assembler Syntax

`MOVE.L (SP+s15),De.E` $\{-2^{14} \leq s15 < 2^{14}, L\}$

`MOVE.L Da.E:Db.E,(SP+s15)` $\{-2^{14} \leq s15 < 2^{14}, L\}$

`MOVE.L (SP+s15),Do.E` $\{-2^{14} \leq s15 < 2^{14}, L\}$

`MOVE.L (a32),De.E` $\{0 \leq a32 < 2^{32}, L\}$

`MOVE.L Da.E:Db.E,(a32)` $\{0 \leq a32 < 2^{32}, L\}$

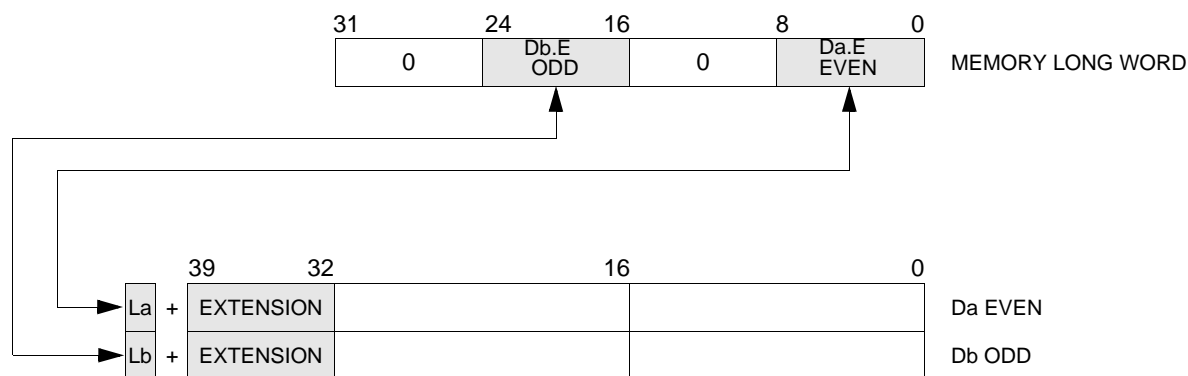
`MOVE.L (a32),Do.E` $\{0 \leq a32 < 2^{32}, L\}$

Description

These six operations save (restore) the extension bits and Ln bit of data registers to (from) memory. One of the operations writes to memory the Ln bit and extension bits of an even and odd pair of registers, as shown below. Another operation reads bits 8:0 from memory to the extension bits and Ln bit of an even register. Another operation reads bits 24:16 to the extension bits and Ln bit of an odd register. The memory address can be specified as an offset to the stack pointer, or as an absolute address.

Writes to memory are done from the even/odd pair of registers. Reads from memory are done to a single register. An extension saved to memory from an even numbered register must be restored to an even register, likewise for odd registers.

The address of the access must be long word-aligned.



Note: Moves of extensions into data registers restore the corresponding limit tag bit (Ln bit) in the destination register.

MOVE.L (SP+s15),De.E

Reads from a memory address pointed to by the stack pointer and a signed 15-bit offset into the extension and Ln bit of an even numbered data register.

MOVE.L Da.E:Db.E,(SP+s15)

Stores the L and extension bits from one even and one odd data register into a 32-bit memory address that is pointed to by the active stack pointer (SP) and a signed 15-bit offset.

MOVE.L (SP+s15),Do.E

Reads from a memory address pointed to by the active stack pointer (SP) and a signed 15-bit offset into the extension and Ln bit of an odd numbered data register.

MOVE.L (a32),De.E

Reads from a 32-bit absolute memory address into the extension and Ln bit of an even numbered data register.

MOVE.L Da.E:Db.E,(a32)

Stores the L and extension bits from one even and one odd data register into a 32-bit absolute memory address.

MOVE.L (a32),Do.E

Reads from a 32-bit absolute memory address into the extension and Ln bit of an odd numbered data register.

Status and Conditions that Affect Instruction

Register Address	Bit Name	Description
SR[18]	EXP	Determines which stack pointer is used when the stack pointer is an operand. Otherwise, the instruction is not affected by SR.
Ln	L	Register to memory moves read the Ln bit with the extension from the source register.

Status and Conditions Changed by Instruction

Register Address	Bit Name	Description
Ln	L	Memory to register moves write the Ln bit in the destination register.

Example

```
move.l d0.e:d1.e,($1224)
```

Register/Memory Address	Before	After
L0:D0	\$1:\$FF FEDC BA98	
L1:D1	\$0:\$00 1234 5678	
\$1224		\$0000 01FF



2

aaaaaaaaaaaaaaaaAAAAAAAAAAAAAAAA

32-bit absolute long address



Operation

Assembler Syntax

(aa) ↔ DR

```
MOVE.L (a32),DR {0 ≤ a32 < 232,L}
MOVE.L DR,(a32)
```

(aa) ↔ C4

```
MOVE.L (a16),C4 {0 ≤ a16 < 216,L}
MOVE.L C4,(a16)
```

(Rn + u3) ↔ DR

```
MOVE.L (Rn+u3),DR {0 ≤ u3 < 32,L}
MOVE.L DR,(Rn+u3)
```

(Rn + s15) ↔ DR

```
MOVE.L (Rn+s15),DR {-214 ≤ s15 < 214,L}
MOVE.L DR,(Rn+s15)
```

(Rn + Rr) ↔ DR

```
MOVE.L (Rn+Rr),DR
MOVE.L DR,(Rn+Rr)
```

(EA) ↔ DR

```
MOVE.L (EA),DR
MOVE.L DR,(EA)
```

(Rn) ↔ C3

```
MOVE.L (Rn),C3
MOVE.L C3,(Rn)
```

(SP – u6) ↔ DR

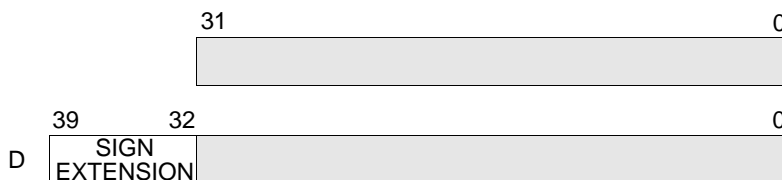
```
MOVE.L (SP-u6),DR {0 ≤ u6 < 256,L}
MOVE.L DR,(SP-u6)
```

(SP + s15) ↔ C4

```
MOVE.L (SP+s15),C4 {-214 ≤ s15 < 214,L}
MOVE.L C4,(SP+s15)
```

Description

These operations move a signed long word (32-bit data) from memory to a register, or from a register to memory. Absolute addresses, offsets, and address register values must be long word-aligned (the address must be a multiple of 4). The programmer should ensure that the effective address resides on a long word boundary.



**MOVE.L (a32),DR****MOVE.L DR,(a32)**

Moves a 32-bit long word between a data or address register and a memory address pointed to by a 32-bit absolute address.

MOVE.L (a16),C4**MOVE.L C4,(a16)**

Moves a 32-bit long word between a general register and a memory address pointed to by a 16-bit unsigned absolute address.

MOVE.L (Rn+u3),DR**MOVE.L DR,(Rn+u3)**

Moves a 32-bit long word between a data or address register and a memory address pointed to by an address register plus a 3-bit unsigned offset that is preshifted right by 2 bits. The offset $u3$, defined by the programmer, must be a multiple of four from 0:28. It is encoded by the assembler with 3 bits, thus creating a 3-bit offset, which is coded in the instruction. The core, when decoding the instruction, post shifts left the 3-bit offset to reconstruct the real offset. This feature enables condensed code size when short immediates are needed.

MOVE.L (Rn+s15),DR**MOVE.L DR,(Rn+s15)**

Moves a 32-bit long word between a data or address register and a memory address pointed to by an address register plus a 15-bit signed offset.

MOVE.L (Rn+Rr),DR**MOVE.L DR,(Rn+Rr)**

Moves a 32-bit long word between a data or address register and a memory address pointed to by an address register plus the contents of a second address register as an offset. The second address register (Rr) is shifted left by 2 bits prior to being added. The modifier mode of this instruction is determined by Rn in MCTL. Rr is limited to $R0$ – $R7$.

MOVE.L (EA),DR**MOVE.L DR,(EA)**

Moves a 32-bit long word between a data or address register and a memory address pointed to by an address register with optional offset or post-increment.

MOVE.L (Rn),C3**MOVE.L C3,(Rn)**

Moves a 32-bit long word between a control, offset, or modifier register and a memory address pointed to by an address register.

MOVE.L (SP-u6),DR**MOVE.L DR,(SP-u6)**

Moves a 32-bit long word between a data or address register and a memory address pointed to by the active stack pointer minus a 6-bit unsigned offset.

MOVE.L (SP+s15),C4

MOVE.L C4,(SP+s15)

Moves a 32-bit long word between a general register and a memory address pointed to by the active stack pointer plus a 15-bit signed offset.

Status and Conditions that Affect Instruction

Register Address	Bit Name	Description
MCTL[31:0]	AM3-AM0	Address modification bits when updating R0–R7. Otherwise, the instruction is not affected by MCTL.
SR[18]	EXP	Determines which stack pointer is used when the stack pointer is an operand. Otherwise, the instruction is not affected by SR.

Status and Conditions Changed by Instruction

Register Address	Bit Name	Description
Ln	L	Clears the Ln bit in the destination register.

Example

```
move.l d0,(r0)
```

Register/Memory Address	Before	After
MCTL	\$0000 0000	
D0	\$FF FFFF FFFA	
R0	\$0000 0084	
\$0084		\$FFFF FFFA

Instruction Formats and Opcodes

Instruction	Words	Cycles	Type	Opcode										
MOVE.L (a32),DR	3	1	3	<table border="1"> <tr> <td>15</td> <td>8</td> <td>7</td> <td>0</td> </tr> <tr> <td>0 0 0 0 H H H H</td> <td>A A A a a w 1 0</td> </tr> <tr> <td>0 0 1 A A A A A</td> <td>A A A A A A A A</td> </tr> <tr> <td>1 0 a a a a a a</td> <td>a a a a a a a a</td> </tr> </table>	15	8	7	0	0 0 0 0 H H H H	A A A a a w 1 0	0 0 1 A A A A A	A A A A A A A A	1 0 a a a a a a	a a a a a a a a
15				8	7	0								
0 0 0 0 H H H H				A A A a a w 1 0										
0 0 1 A A A A A	A A A A A A A A													
1 0 a a a a a a	a a a a a a a a													
MOVE.L DR,(a32)														
MOVE.L (a16),C4	2	1	3	<table border="1"> <tr> <td>15</td> <td>8</td> <td>7</td> <td>0</td> </tr> <tr> <td>0 0 0 w D D D D</td> <td>A A A 0 1 0 D 1</td> </tr> <tr> <td>1 0 0 A A A A A</td> <td>A A A A A A A A</td> </tr> </table>	15	8	7	0	0 0 0 w D D D D	A A A 0 1 0 D 1	1 0 0 A A A A A	A A A A A A A A		
15				8	7	0								
0 0 0 w D D D D	A A A 0 1 0 D 1													
1 0 0 A A A A A	A A A A A A A A													
MOVE.L C4,(a16)														
MOVE.L (Rn+u3),DR	1	2	4	<table border="1"> <tr> <td>15</td> <td>8</td> <td>7</td> <td>0</td> </tr> <tr> <td>1 0 1 1 H H H H</td> <td>w 1 R R R s s s</td> </tr> </table>	15	8	7	0	1 0 1 1 H H H H	w 1 R R R s s s				
15	8	7	0											
1 0 1 1 H H H H	w 1 R R R s s s													

MOVE.L DR, (Rn+u3)				15	8	7	0
MOVE.L (Rn+s15),DR	2	2	3	0	0	0	w H H H H 1 s s 0 0 R R R
MOVE.L DR, (Rn+s15)				1	0	0	s s s s s s s s s s
MOVE.L (Rn+Rr),DR				15	8	7	0
MOVE.L (Rn+Rr),DR	1	2	4	1	0	1	0 H H H H w 1 R R R r r r
MOVE.L DR, (Rn+Rr)							
MOVE.L (EA),DR				15	8	7	0
MOVE.L (EA),DR	1	1 ¹	1	0	*	0	w H H H H 1 0 M M M R R R
MOVE.L DR, (EA)							
MOVE.L (Rn),C3				15	8	7	0
MOVE.L (Rn),C3	1	1	4	1	0	0	1 D D D D 0 0 0 1 w R R R
MOVE.L C3, (Rn)							
MOVE.L (SP-u6),DR				15	8	7	0
MOVE.L (SP-u6),DR	1	2	2	1	1	1	1 H H H H w 1 s s s s s s
MOVE.L DR, (SP-u6)							
MOVE.L (SP+s15),C4				15	8	7	0
MOVE.L (SP+s15),C4	2	2	3	0	0	0	w D D D D 1 s s 1 1 0 D 0
MOVE.L C4, (SP+s15)				1	0	0	s s s s s s s s s s

Note: ** indicates serial grouping encoding.

Note 1: When the form (Rn + N0) is used in EA, the cycle count is increased by 1.



struction Fields

C3 DDDD General Registers

0000	B0	0100	B4	1000	N0	1100	M0
0001	B1	0101	B5	1001	N1	1101	M1
0010	B2	0110	B6	1010	N2	1110	M2
0011	B3	0111	B7	1011	N3	1111	M3

C4 DDDDD General Registers

00000	D0	01000	D4	10000	R0	11000	R4
00001	B0	01001	B4	10001	N0	11001	M0
00010	D1	01010	D5	10010	R1	11010	R5
00011	B1	01011	B5	10011	N1	11011	M1
00100	D2	01100	D6	10100	R2	11100	R6
00101	B2	01101	B6	10101	N2	11101	M2
00110	D3	01110	D7	10110	R3	11110	R7
00111	B3	01111	B7	10111	N3	11111	M3

Note: This instruction can specify D8-D15 or R8-R15 as operands by using a high register prefix.

DR HHHH Data/Address Register

0000	D0	0100	D4	1000	R0	1100	R4
0001	D1	0101	D5	1001	R1	1101	R5
0010	D2	0110	D6	1010	R2	1110	R6
0011	D3	0111	D7	1011	R3	1111	R7

Note: This instruction can specify D8-D15 or R8-R15 as operands by using a high register prefix.

EA MMM Effective Address Notation

000	(Rn+N0)	010	(Rn)	100	(Rn)+N0	110	(Rn)+N2
001	(Rn)–	011	(Rn)+	101	(Rn)+N1	111	(Rn)+N3

Rn RRR Address Register

000	R0	010	R2	100	R4	110	R6
001	R1	011	R3	101	R5	111	R7

Note: This instruction can specify R8-R15 as operands by using a high register prefix.

Rr rrr Address Register

000	R0	010	R2	100	R4	110	R6
001	R1	011	R3	101	R5	111	R7

Note: The Rr operand is limited to R0–R7.

w Read/Write Notation

0	write	1	read
---	-------	---	------



a16	AAAAAAAAAAAAAAAA	16-bit unsigned absolute address
a32	aaaaaaaaaaaaaaaaAAAAAAAAAAAAAAAA	32-bit absolute long address
s15	ssssssssssssss	Signed 15-bit offset
u3	sss00	Unsigned 3-bit offset
u6	sssss00	Unsigned 6-bit SP offset

MOVE.W Move Immediate Integer Word (AGU) MOVE.W

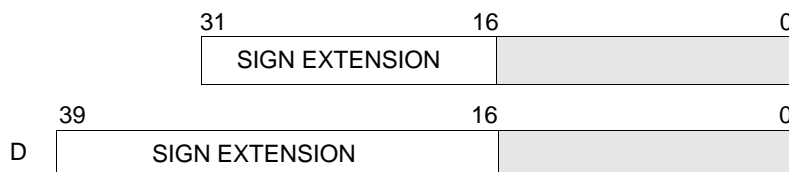
Operation

Assembler Syntax

#s7 → DR	MOVE.W #s7,DR $\{-64 \leq s7 < 64\}$
#s16 → C4	MOVE.W #s16,C4 $\{-2^{15} \leq s16 < 2^{15}\}$
#s16 → (aa)	MOVE.W #s16,(a16) $\{-2^{15} \leq s16 < 2^{15}\}\{0 \leq a16 < 2^{16},W\}$
#s16 → (SP-u5)	MOVE.W #s16,(SP-u5) $\{-2^{15} \leq s16 < 2^{15}\}\{0 \leq u5 < 64,W\}$
#s16 → (Rn)	MOVE.W #s16,(Rn) $\{-2^{15} \leq s16 < 2^{15}\}$
#s16 → (SP+sa16)	MOVE.W #s16,(SP+sa16) $\{-2^{15} \leq s16 < 2^{15}\}\{-2^{15} \leq sa16 < 2^{15},W\}$

Description

These operations move a signed immediate integer word to a register or a memory address. The address of the access must be word-aligned.



MOVE.W #s7,DR

Loads an immediate signed 7-bit value into the LP of a data or address register and sign-extends it.

MOVE.W #s16,C4

Loads an immediate signed 16-bit value into the LP of a general register and sign-extends it.

MOVE.W #s16,(a16)

Writes an immediate signed 16-bit value to an absolute 16-bit address.

MOVE.W #s16,(SP-u5)

Writes an immediate signed 16-bit value to a memory address pointed to by the active stack pointer (SP) minus an unsigned 5-bit offset that is preshifted left 1 bit.

MOVE.W #s16,(Rn)

Writes an immediate signed 16-bit value to a memory address pointed to by an address register.

MOVE.W

#s16,(SP+sa16)

Writes a 16-bit signed immediate value to a memory address pointed to by the active stack pointer (SP) plus a signed 16-bit offset.



Status and Conditions that Affect Instruction

Register Address	Bit Name	Description
SR[18]	EXP	Determines which stack pointer is used when the stack pointer is an operand. Otherwise, the instruction is not affected by SR.

Status and Conditions Changed by Instruction

Register Address	Bit Name	Description
Ln	L	Clears the Ln bit in the destination register.

Example

```
move.w #$0050,r7
```

Register/Memory Address	Before	After
immediate	<input type="text" value="\$0000 0050"/>	
R7		<input type="text" value="\$0000 0050"/>



Instruction Formats and Opcodes

Instruction	Words	Cycles	Type	Opcode																
MOVE.W #s7,DR	1	1	2	<table border="1"> <tr> <td>15</td> <td>8</td> <td>7</td> <td>0</td> </tr> <tr> <td>1 1 0 0 H H H H</td> <td>1</td> <td>i i i i i i i i</td> <td></td> </tr> </table>	15	8	7	0	1 1 0 0 H H H H	1	i i i i i i i i									
15	8	7	0																	
1 1 0 0 H H H H	1	i i i i i i i i																		
MOVE.W #s16,C4	2	1	4	<table border="1"> <tr> <td>15</td> <td>8</td> <td>7</td> <td>0</td> </tr> <tr> <td>0 0 1 0 D D D D</td> <td>i i i 0 0 0 D 0</td> <td></td> <td></td> </tr> <tr> <td>1 0 0 i i i i i</td> <td>i i i i i i i i</td> <td></td> <td></td> </tr> </table>	15	8	7	0	0 0 1 0 D D D D	i i i 0 0 0 D 0			1 0 0 i i i i i	i i i i i i i i						
15	8	7	0																	
0 0 1 0 D D D D	i i i 0 0 0 D 0																			
1 0 0 i i i i i	i i i i i i i i																			
MOVE.W #s16,(a16)	3	1	3	<table border="1"> <tr> <td>15</td> <td>8</td> <td>7</td> <td>0</td> </tr> <tr> <td>0 0 1 1 1 0 0 0</td> <td>A A A i i 1 0 0</td> <td></td> <td></td> </tr> <tr> <td>0 0 1 A A A A A</td> <td>A A A A A A A A</td> <td></td> <td></td> </tr> <tr> <td>1 0 i i i i i i</td> <td>i i i i i i i i</td> <td></td> <td></td> </tr> </table>	15	8	7	0	0 0 1 1 1 0 0 0	A A A i i 1 0 0			0 0 1 A A A A A	A A A A A A A A			1 0 i i i i i i	i i i i i i i i		
15	8	7	0																	
0 0 1 1 1 0 0 0	A A A i i 1 0 0																			
0 0 1 A A A A A	A A A A A A A A																			
1 0 i i i i i i	i i i i i i i i																			
MOVE.W #s16,(SP-u5)	2	2	3	<table border="1"> <tr> <td>15</td> <td>8</td> <td>7</td> <td>0</td> </tr> <tr> <td>0 0 0 1 1 0 0 0</td> <td>i i i A A A A A</td> <td></td> <td></td> </tr> <tr> <td>1 0 1 i i i i i</td> <td>i i i i i i i i</td> <td></td> <td></td> </tr> </table>	15	8	7	0	0 0 0 1 1 0 0 0	i i i A A A A A			1 0 1 i i i i i	i i i i i i i i						
15	8	7	0																	
0 0 0 1 1 0 0 0	i i i A A A A A																			
1 0 1 i i i i i	i i i i i i i i																			
MOVE.W #s16,(Rn)	2	1	3	<table border="1"> <tr> <td>15</td> <td>8</td> <td>7</td> <td>0</td> </tr> <tr> <td>0 0 0 1 1 0 0 1</td> <td>i i i 0 1 R R R</td> <td></td> <td></td> </tr> <tr> <td>1 0 1 i i i i i</td> <td>i i i i i i i i</td> <td></td> <td></td> </tr> </table>	15	8	7	0	0 0 0 1 1 0 0 1	i i i 0 1 R R R			1 0 1 i i i i i	i i i i i i i i						
15	8	7	0																	
0 0 0 1 1 0 0 1	i i i 0 1 R R R																			
1 0 1 i i i i i	i i i i i i i i																			
MOVE.W #s16,(SP+sa16)	3	2	3	<table border="1"> <tr> <td>15</td> <td>8</td> <td>7</td> <td>0</td> </tr> <tr> <td>0 0 1 1 1 0 0 0</td> <td>A A A i i 1 0 1</td> <td></td> <td></td> </tr> <tr> <td>0 0 1 A A A A A</td> <td>A A A A A A A A</td> <td></td> <td></td> </tr> <tr> <td>1 0 i i i i i i</td> <td>i i i i i i i i</td> <td></td> <td></td> </tr> </table>	15	8	7	0	0 0 1 1 1 0 0 0	A A A i i 1 0 1			0 0 1 A A A A A	A A A A A A A A			1 0 i i i i i i	i i i i i i i i		
15	8	7	0																	
0 0 1 1 1 0 0 0	A A A i i 1 0 1																			
0 0 1 A A A A A	A A A A A A A A																			
1 0 i i i i i i	i i i i i i i i																			

Instruction Fields

C4	DDDDD	General Registers					
00000	D0	01000	D4	10000	R0	11000	R4
00001	B0	01001	B4	10001	N0	11001	M0
00010	D1	01010	D5	10010	R1	11010	R5
00011	B1	01011	B5	10011	N1	11011	M1
00100	D2	01100	D6	10100	R2	11100	R6
00101	B2	01101	B6	10101	N2	11101	M2
00110	D3	01110	D7	10110	R3	11110	R7
00111	B3	01111	B7	10111	N3	11111	M3

Note: This instruction can specify D8-D15 or R8-R15 as operands by using a high register prefix.

HHHH Data/Address Register

0000	D0	0100	D4	1000	R0	1100	R4
0001	D1	0101	D5	1001	R1	1101	R5
0010	D2	0110	D6	1010	R2	1110	R6
0011	D3	0111	D7	1011	R3	1111	R7

Note: This instruction can specify D8-D15 or R8-R15 as operands by using a high register prefix.

Rn RRR Address Register

000	R0	010	R2	100	R4	110	R6
001	R1	011	R3	101	R5	111	R7

Note: This instruction can specify R8-R15 as operands by using a high register prefix.

#s7	iiiiiii	7-bit signed immediate data
#s16	iiiiiiiiiiiiiii	16-bit signed immediate data
a16	AAAAAAAAAAAAAAAA	16-bit unsigned absolute address
sa16	AAAAAAAAAAAAAAAA	Signed 16-bit offset
u5	AAAAA0	Unsigned 5-bit SP offset

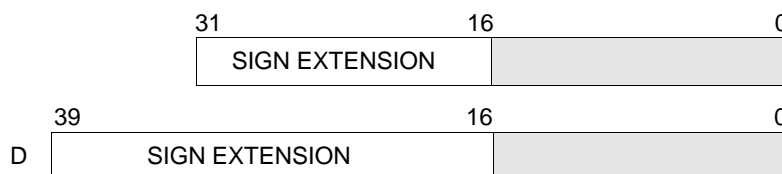
MOVE.W**Move Integer Word (AGU)****MOVE.W****Operation****Assembler Syntax**

(aa) ↔ DR	MOVE.W (a32),DR {0 ≤ a32 < 2 ³² ,W} MOVE.W DR,(a32)
(aa) ↔ C4	MOVE.W (a16),C4 {0 ≤ a16 < 2 ¹⁶ ,W} MOVE.W C4,(a16)
(Rn+u3) ↔ DR	MOVE.W (Rn+u3),DR {0 ≤ u3 < 16,W} MOVE.W DR,(Rn+u3)
(Rn+s15) ↔ DR	MOVE.W (Rn+s15),DR {-2 ¹⁴ ≤ s15 < 2 ¹⁴ ,W} MOVE.W DR,(Rn+s15)
(Rn+Rr) ↔ DR	MOVE.W (Rn+Rr),DR MOVE.W DR,(Rn+Rr)
(EA) ↔ DR	MOVE.W (EA),DR MOVE.W DR,(EA)
(Rn) ↔ C3	MOVE.W (Rn),C3 MOVE.W C3,(Rn)
(SP-u6) ↔ DR	MOVE.W (SP-u6),DR {0 ≤ u6 < 128,W} MOVE.W DR,(SP-u6)
(SP+s15) ↔ C4	MOVE.W (SP+s15),C4 {-2 ¹⁴ ≤ s15 < 2 ¹⁴ ,W} MOVE.W C4,(SP+s15)

Description

These operations either read a signed integer word from memory into the LP of a register and sign-extend it, or write a signed integer word from the LP of a register to a memory address.

The address of the access must be word-aligned.

**MOVE.W (a32),DR****MOVE.W DR,(a32)**

Moves a signed word between a data or address register (DR) and an absolute 32-bit address.

MOVE.W (a16),C4**MOVE.W C4,(a16)**

Moves a signed word between a general register (C4) and an absolute 16-bit address.

**MOVE.W (Rn+u3),DR****MOVE.W DR,(Rn+u3)**

Moves a signed word between a data or address register (DR) and a memory address pointed to by an address register (Rn) with an unsigned 3-bit offset that is preshifted right by 1 bit. The offset u3, defined by the programmer, must be an even integer from 0–14. It is encoded by the assembler with 3 bits, thus creating a 3-bit offset, which is coded in the instruction. The core, when decoding the instruction, post shifts left the 3-bit offset to reconstruct the real offset. This feature enables condensed code size when short immediates are needed.

MOVE.W (Rn+s15),DR**MOVE.W DR,(Rn+s15)**

Moves a signed word between a data or address register (DR) and a memory address pointed to by an address register (Rn) with a signed 15-bit offset.

MOVE.W (Rn+Rr),DR**MOVE.W DR,(Rn+Rr)**

Moves a signed word between a data or address register (DR) and a memory address pointed to by an address register (Rn) with an offset contained in another address register (Rr). The second address register (Rr) is shifted left by one bit prior to being added. The modifier mode of this instruction is determined by Rn in MCTL. Rr is limited to R0–R7.

MOVE.W (EA),DR**MOVE.W DR,(EA)**

Moves a signed word between a data or address register (DR) and a memory address pointed to by (EA) an address register with an optional offset or post-increment.

MOVE.W (Rn),C3**MOVE.W C3,(Rn)**

Moves a signed word between a control, offset, or modifier register (C3) and a memory address pointed to by an address register (Rn).

MOVE.W (SP–u6),DR**MOVE.W DR,(SP–u6)**

Moves a signed word between a data or address register (DR) and a memory address pointed to by the active stack pointer (SP) minus a 6-bit unsigned offset.

MOVE.W (SP+s15),C4**MOVE.W C4,(SP+s15)**

Moves a signed word between a general register (C4) and a memory address pointed to by the active stack pointer (SP) with a signed 15-bit offset.

Status and Conditions that Affect Instruction

Register Address	Bit Name	Description
MCTL[31:0]	AM3–AM0	Address modification bits when updating R0–R7. Otherwise, the instruction is not affected by MCTL.
SR[18]	EXP	Determines which stack pointer is used when the stack pointer is an operand. Otherwise, the instruction is not affected by SR.

Status and Conditions Changed by Instruction

Register Address	Bit Name	Description
Ln	L	Clears the Ln bit in the destination registers.

Example

```
move.w d1, (r7+4)
```

Register/Memory Address	Before	After
MCTL	\$0000 0000	
d1	\$FF FFFF FFF1	
R7	\$0000 000A	\$0000 000A
\$000E		\$FFF1

Instruction Formats and Opcodes

Instruction	Words	Cycles	Type	Opcode																
MOVE.W (a32),DR MOVE.W DR,(a32)	3	1	3	<table border="1"> <tr> <td>15</td> <td>8</td> <td>7</td> <td>0</td> </tr> <tr> <td>0 0 0 0 H H H H</td> <td>A A A a a w 0 0</td> <td></td> <td></td> </tr> <tr> <td>0 0 1 A A A A A</td> <td>A A A A A A A A</td> <td></td> <td></td> </tr> <tr> <td>1 0 a a a a a a</td> <td>a a a a a a a a</td> <td></td> <td></td> </tr> </table>	15	8	7	0	0 0 0 0 H H H H	A A A a a w 0 0			0 0 1 A A A A A	A A A A A A A A			1 0 a a a a a a	a a a a a a a a		
15	8	7	0																	
0 0 0 0 H H H H	A A A a a w 0 0																			
0 0 1 A A A A A	A A A A A A A A																			
1 0 a a a a a a	a a a a a a a a																			
MOVE.W (a16),C4 MOVE.W C4,(a16)	2	1	3	<table border="1"> <tr> <td>15</td> <td>8</td> <td>7</td> <td>0</td> </tr> <tr> <td>0 0 0 w D D D D</td> <td>A A A 0 1 0 D 0</td> <td></td> <td></td> </tr> <tr> <td>1 0 0 A A A A A</td> <td>A A A A A A A A</td> <td></td> <td></td> </tr> </table>	15	8	7	0	0 0 0 w D D D D	A A A 0 1 0 D 0			1 0 0 A A A A A	A A A A A A A A						
15	8	7	0																	
0 0 0 w D D D D	A A A 0 1 0 D 0																			
1 0 0 A A A A A	A A A A A A A A																			
MOVE.W (Rn+u3),DR MOVE.W DR,(Rn+u3)	1	2	4	<table border="1"> <tr> <td>15</td> <td>8</td> <td>7</td> <td>0</td> </tr> <tr> <td>1 0 1 1 H H H H</td> <td>w 0 R R R s s s</td> <td></td> <td></td> </tr> </table>	15	8	7	0	1 0 1 1 H H H H	w 0 R R R s s s										
15	8	7	0																	
1 0 1 1 H H H H	w 0 R R R s s s																			
MOVE.W (Rn+s15),DR MOVE.W DR,(Rn+s15)	2	2	3	<table border="1"> <tr> <td>15</td> <td>8</td> <td>7</td> <td>0</td> </tr> <tr> <td>0 0 0 w H H H H</td> <td>0 s s 0 0 R R R</td> <td></td> <td></td> </tr> <tr> <td>1 0 0 s s s s s</td> <td>s s s s s s s s</td> <td></td> <td></td> </tr> </table>	15	8	7	0	0 0 0 w H H H H	0 s s 0 0 R R R			1 0 0 s s s s s	s s s s s s s s						
15	8	7	0																	
0 0 0 w H H H H	0 s s 0 0 R R R																			
1 0 0 s s s s s	s s s s s s s s																			
MOVE.W (Rn+Rr),DR MOVE.W DR,(Rn+Rr)	1	2	4	<table border="1"> <tr> <td>15</td> <td>8</td> <td>7</td> <td>0</td> </tr> <tr> <td>1 0 1 0 H H H H</td> <td>w 0 R R R r r r</td> <td></td> <td></td> </tr> </table>	15	8	7	0	1 0 1 0 H H H H	w 0 R R R r r r										
15	8	7	0																	
1 0 1 0 H H H H	w 0 R R R r r r																			
MOVE.W (EA),DR MOVE.W DR,(EA)	1	1 ²	1	<table border="1"> <tr> <td>15</td> <td>8</td> <td>7</td> <td>0</td> </tr> <tr> <td>0 * 0 w H H H H</td> <td>0 0 M M M R R R</td> <td></td> <td></td> </tr> </table>	15	8	7	0	0 * 0 w H H H H	0 0 M M M R R R										
15	8	7	0																	
0 * 0 w H H H H	0 0 M M M R R R																			
Notes: <ol style="list-style-type: none"> ** indicates serial grouping encoding. When the form (Rn + N0) is used in EA, the cycle count is increased by 1. 																				
MOVE.W (Rn),C3 MOVE.W C3,(Rn)	1	1	4	<table border="1"> <tr> <td>15</td> <td>8</td> <td>7</td> <td>0</td> </tr> <tr> <td>1 0 0 1 D D D D</td> <td>0 0 0 0 w R R R</td> <td></td> <td></td> </tr> </table>	15	8	7	0	1 0 0 1 D D D D	0 0 0 0 w R R R										
15	8	7	0																	
1 0 0 1 D D D D	0 0 0 0 w R R R																			
MOVE.W (SP+s15),C4 MOVE.W C4,(SP+s15)	2	2	3	<table border="1"> <tr> <td>15</td> <td>8</td> <td>7</td> <td>0</td> </tr> <tr> <td>0 0 0 w D D D D</td> <td>0 s s 1 1 0 D 0</td> <td></td> <td></td> </tr> <tr> <td>1 0 0 s s s s s</td> <td>s s s s s s s s</td> <td></td> <td></td> </tr> </table>	15	8	7	0	0 0 0 w D D D D	0 s s 1 1 0 D 0			1 0 0 s s s s s	s s s s s s s s						
15	8	7	0																	
0 0 0 w D D D D	0 s s 1 1 0 D 0																			
1 0 0 s s s s s	s s s s s s s s																			
MOVE.W (SP-u6),DR MOVE.W DR,(SP-u6)	1	2	2	<table border="1"> <tr> <td>15</td> <td>8</td> <td>7</td> <td>0</td> </tr> <tr> <td>1 1 1 1 H H H H</td> <td>w 0 s s s s s s</td> <td></td> <td></td> </tr> </table>	15	8	7	0	1 1 1 1 H H H H	w 0 s s s s s s										
15	8	7	0																	
1 1 1 1 H H H H	w 0 s s s s s s																			



struction Fields

w Read/Write Notation

0	write	1	read
---	-------	---	------

C3 DDDD General Registers

0000	B0	0100	B4	1000	N0	1100	M0
0001	B1	0101	B5	1001	N1	1101	M1
0010	B2	0110	B6	1010	N2	1110	M2
0011	B3	0111	B7	1011	N3	1111	M3

C4 DDDDD General Registers

00000	D0	01000	D4	10000	R0	11000	R4
00001	B0	01001	B4	10001	N0	11001	M0
00010	D1	01010	D5	10010	R1	11010	R5
00011	B1	01011	B5	10011	N1	11011	M1
00100	D2	01100	D6	10100	R2	11100	R6
00101	B2	01101	B6	10101	N2	11101	M2
00110	D3	01110	D7	10110	R3	11110	R7
00111	B3	01111	B7	10111	N3	11111	M3

Note: This instruction can specify D8-D15 or R8-R15 as operands by using a high register prefix.

DR HHHH Data/Address Register

0000	D0	0100	D4	1000	R0	1100	R4
0001	D1	0101	D5	1001	R1	1101	R5
0010	D2	0110	D6	1010	R2	1110	R6
0011	D3	0111	D7	1011	R3	1111	R7

Note: This instruction can specify D8-D15 or R8-R15 as operands by using a high register prefix.

EA MMM Effective Address Notation

000	(Rn+N0)	010	(Rn)	100	(Rn)+N0	110	(Rn)+N2
001	(Rn)–	011	(Rn)+	101	(Rn)+N1	111	(Rn)+N3

Rn RRR Address Register

000	R0	010	R2	100	R4	110	R6
001	R1	011	R3	101	R5	111	R7

Note: This instruction can specify R8-R15 as operands by using a high register prefix.

Rr rrr Address Register

000	R0	010	R2	100	R4	110	R6
001	R1	011	R3	101	R5	111	R7

Note: The Rr register file is limited to the lower bank R0–R7.



a16	AAAAAAAAAAAAAAAA	16-bit unsigned absolute address
a32	aaaaaaaaaaaaaaaaAAAAAAAAAAAAAAAA	32-bit absolute long address
s15	ssssssssssssss	Signed 15-bit offset
u3	sss0	Unsigned 3-bit offset
u6	sssss0	Unsigned 6-bit SP offset

MOVEc Conditional Address Register Move (AGU) MOVEc

Operation

If T=1, then Rq → Rn

If T=0, then Rq → Rn

Assembler Syntax

MOVET Rq,Rn

MOVEF Rq,Rn

Description

This instruction conditionally copies the value of one address register to another, depending on the value of the T bit in SR. These operations have the same timing as other move instructions. MOVEc is performed in the execution stage of the pipeline, unlike TFRA, which is performed in the address generation stage.

MOVET Rq,Rn

Copies one address register to another if the T bit is set.

MOVEF Rq,Rn

Copies one address register to another if the T bit is cleared.

Status and Conditions that Affect Instruction

Register Address	Bit Name	Description
SR[1]	T	True bit

Status and Conditions Changed by Instruction

None.

Example

```
movet r0,r1
```

Register/Memory Address	Before	After
SR	\$00E4 0002	
R0	\$0000 0010	
R1		\$0000 0010

Note: \$00E4 0002 in the status register indicates that the true bit is set.



Instruction Formats and Opcodes

Instruction	Words	Cycles	Type	Opcode																
MOVET Rq, Rn	1	1	4	<div style="display: flex; justify-content: space-between; margin-bottom: 5px;"> 15 8 7 0 </div> <table border="1" style="width: 100%; border-collapse: collapse;"> <tr> <td style="width: 10%;">1</td> <td style="width: 10%;">0</td> <td style="width: 10%;">0</td> <td style="width: 10%;">1</td> <td style="width: 10%;">1</td> <td style="width: 10%;">R</td> <td style="width: 10%;">R</td> <td style="width: 10%;">R</td> <td style="width: 10%;">0</td> <td style="width: 10%;">1</td> <td style="width: 10%;">0</td> <td style="width: 10%;">1</td> <td style="width: 10%;">0</td> <td style="width: 10%;">q</td> <td style="width: 10%;">q</td> <td style="width: 10%;">q</td> </tr> </table>	1	0	0	1	1	R	R	R	0	1	0	1	0	q	q	q
1	0	0	1	1	R	R	R	0	1	0	1	0	q	q	q					
MOVEF Rq, Rn	1	1	4	<div style="display: flex; justify-content: space-between; margin-bottom: 5px;"> 15 8 7 0 </div> <table border="1" style="width: 100%; border-collapse: collapse;"> <tr> <td style="width: 10%;">1</td> <td style="width: 10%;">0</td> <td style="width: 10%;">0</td> <td style="width: 10%;">1</td> <td style="width: 10%;">1</td> <td style="width: 10%;">R</td> <td style="width: 10%;">R</td> <td style="width: 10%;">R</td> <td style="width: 10%;">0</td> <td style="width: 10%;">1</td> <td style="width: 10%;">0</td> <td style="width: 10%;">1</td> <td style="width: 10%;">1</td> <td style="width: 10%;">q</td> <td style="width: 10%;">q</td> <td style="width: 10%;">q</td> </tr> </table>	1	0	0	1	1	R	R	R	0	1	0	1	1	q	q	q
1	0	0	1	1	R	R	R	0	1	0	1	1	q	q	q					

Instruction Fields

Rq	qqq	Address Register					
000	R0	010	R2	100	R4	110	R6
001	R1	011	R3	101	R5	111	R7

Note: This instruction can specify R8-R15 as operands by using a high register prefix.

Rn	RRR	Address Register					
000	R0	010	R2	100	R4	110	R6
001	R1	011	R3	101	R5	111	R7

Note: This instruction can specify R8-R15 as operands by using a high register prefix.



MOVES.2F Move Two Fractional Words to Memory With Scaling and Saturation (AGU) MOVES.2F

Operation

Da:Db → (EA)

Assembler Syntax

MOVES.2F Da:Db, (EA)

Description

The data that is moved from each register to memory is scaled according to the scaling mode. If the Ln bit is set, the moved data is also saturated. The address register values must be long aligned. This instruction is affected by SM (Saturation Mode bit - SR[2]). When SM is set, scaling is not performed, and the scale bits S[1:10] have no effect.

MOVES.2F Da:Db,(EA)

Moves two signed fractional words from a data register pair to a memory address pointed to by an address register with an optional offset or post-increment.

The first operand (Da) will be moved to the lower memory address (EA). The second operand (Db) will be moved to memory address (EA + 2). In order to keep this behavior in both big endian and little endian modes, the core will interpret the data bus differently in each mode. See [Section 2.4.1, “SC140 Endian Support,”](#) on page 2-56, for more detail on bus and memory behavior for each mode.

Status and Conditions that Affect Instruction

Register Address	Bit Name	Description
MCTL[31:0]	AM3–AM0	Address modification bits when updating R0–R7. Otherwise, the instruction is not affected by MCTL.
SR[5:4]	S[1:0]	Scaling mode bits choose: no scaling, scale up one bit, or scale down one bit.
Ln	L	Limited values are written to the destination if the Ln bit is set.
EMR[16]	BEM	Set if big endian mode, cleared if little endian mode.

Status and Conditions Changed by Instruction

Register Address	Bit Name	Description
SR[6]	S	Scaling bit, set when the absolute value of either or both of the words moved (after scaling and limiting) is greater than or equal to 0.25 and less than 0.75.

Example

```
moves.2f d0:d1, (r0)
```

Register/Memory Address	Before	After
MCTL		\$0000 0000

Register/Memory Address	Before	After
SR	\$00E0 0000	\$00E0 0000
d0	\$1:\$00 8000 0000	
d1	\$0:\$00 7EAC F00D	
R0	\$0000 0054	
\$0054		\$7FFF
\$0056		\$7EAC

The Ln bit is set in d0, and the number in d0 is positive (bit 39 = 0), so the saturated value \$7FFF is written to memory.

Instruction Formats and Opcodes

Instruction	Words	Cycles	Type	Opcode																																
MOVES.2F Da:Db, (EA)	1	1 ²	1	<table border="1" style="width: 100%; text-align: center;"> <tr> <td colspan="4">15</td> <td colspan="4">8</td> <td colspan="4">7</td> <td colspan="4">0</td> </tr> <tr> <td>0</td><td>*</td><td>0</td><td>0</td> <td>1</td><td>h</td><td>h</td><td>1</td> <td>0</td><td>1</td> <td>M</td><td>M</td> <td>M</td><td>R</td> <td>R</td><td>R</td> </tr> </table>	15				8				7				0				0	*	0	0	1	h	h	1	0	1	M	M	M	R	R	R
15				8				7				0																								
0	*	0	0	1	h	h	1	0	1	M	M	M	R	R	R																					

- Notes:**
- ** indicates serial grouping encoding.
 - When the form (Rn + N0) is used in EA, the cycle count is increased by 1.

Instruction Fields

Da:Db	hh	Data Register Pairs					
00	D0:D1	01	D2:D3	10	D4:D5	11	D6:D7

Note: This instruction can specify D8-D15 as operands by using a prefix.

EA	MMM	Effective Address Notation					
000	(Rn+N0)	010	(Rn)	100	(Rn)+N0	110	(Rn)+N2
001	(Rn)-	011	(Rn)+	101	(Rn)+N1	111	(Rn)+N3

Rn	RRR	Address Register					
000	R0	010	R2	100	R4	110	R6
001	R1	011	R3	101	R5	111	R7

Note: This instruction can specify R8-R15 as operands by using a high register prefix.



MOVES.4F Move Four Fractional Words to Memory With Scaling and Saturation (AGU) MOVES.4F

Operation

Da:Db:Dc:Dd → (EA)

Assembler Syntax

MOVES.4F Da:Db:Dc:Dd, (EA)

Description

The data that is moved from each register to memory is scaled according to the scaling mode. If the Ln bit is set, it is also saturated. The address register values must be quad word-aligned (a multiple of 8). This instruction is affected by SM (Saturation Mode bit - SR[2]). When SM is set, scaling is not performed, and the scale bits S[1:0] have no effect.

MOVES.4F Da:Db:Dc:Dd,(EA)

Writes four signed fractional words from a data register quad to memory addresses pointed to by an address register with an optional offset or post-increment.

The first operand (Da) will be moved to the lower memory address (EA). The second operand (Db) will be moved to memory address (EA + 2). The third operand (Dc) will be moved to memory address (EA + 4). And, the fourth operand (Dd) will be moved to memory address (EA + 6). In order to keep this behavior in both big endian and little endian modes, the core will drive the data bus differently in each mode. See [Section 2.4.1, “SC140 Endian Support,”](#) on page 2-56, for more detail on bus and memory behavior for each mode.

Status and Conditions that Affect Instruction

Register Address	Bit Name	Description
MCTL[31:0]	AM3–AM0	Address modification bits when updating R0–R7. Otherwise, the instruction is not affected by MCTL.
SR[5:4]	S[1:0]	Scaling mode bits choose: no scaling, scale up one bit, or scale down one bit.
Ln	L	Limited values are written to the destination if the Ln bit is set.
EMR[16]	BEM	Set if big endian mode, cleared if little endian mode.

Status and Conditions Changed by Instruction

Register Address	Bit Name	Description
SR[6]	S	Scaling bit, set when the absolute value of any one of the words moved (after scaling and limiting) is greater than or equal to 0.25 and less than 0.75.

Example

moves.4f d0:d1:d2:d3, (r0)

Register/Memory Address	Before	After
SR		\$00E0 0000

Register/Memory Address	Before	After
R0	\$0000 0050	
L0:D0	\$1:\$0 08000 0000	
L1:D1	\$0:\$00 7FFF FFFF	
L2:D2	\$1:\$87 6543 2100	
L3:D3	\$0:\$FF 8765 4321	
\$0050		\$7FFF
\$0052		\$7FFF
\$0054		\$8000
\$0056		\$8765

Instruction Formats and Opcodes

Instruction	Words	Cycles	Type	Opcode																																
MOVES.4F	1	1 ²	1	<table border="1" style="width: 100%; text-align: center;"> <tr> <td colspan="4">15</td> <td colspan="4">8</td> <td colspan="4">7</td> <td colspan="4">0</td> </tr> <tr> <td>0</td><td>*</td><td>0</td><td>0</td><td>1</td><td>k</td><td>0</td><td>0</td> <td>1</td><td>1</td><td>M</td><td>M</td><td>M</td><td>R</td><td>R</td><td>R</td> </tr> </table>	15				8				7				0				0	*	0	0	1	k	0	0	1	1	M	M	M	R	R	R
15				8				7				0																								
0	*	0	0	1	k	0	0	1	1	M	M	M	R	R	R																					

Da:Db:Dc:Dd, (EA)

- Notes:**
- ** indicates serial grouping encoding.
 - When the form (Rn + N0) is used in EA, the cycle count is increased by 1.

Instruction Fields

Da:Db:Dc:Dd	k	Data Register Quad			
0		D0:D1:D2:D3	1	D4:D5:D6:D7	

Note: This instruction can specify D8-D15 as operands by using a prefix.

EA	MMM Effective Address Notation						
000	(Rn+N0)	010	(Rn)	100	(Rn)+N0	110	(Rn)+N2
001	(Rn)–	011	(Rn)+	101	(Rn)+N1	111	(Rn)+N3

Rn	RRR Address Register						
000	R0	010	R2	100	R4	110	R6
001	R1	011	R3	101	R5	111	R7

Note: This instruction can specify R8-R15 as operands by using a high register prefix.



MOVES.F

Move Fractional Word to Memory With Scaling and Saturation (AGU)

MOVES.F

Operation

Db → (aa)

Db → (aa)

Db → (Rn + s15)

Db → (EA)

Db → (SP + s15)

Assembler Syntax

MOVES.F Db, (a16) { $0 \leq a16 < 2^{16}, W$ }

MOVES.F Db, (a32) { $0 \leq a32 < 2^{32}, W$ }

MOVES.F Db, (Rn+s15) { $-2^{14} \leq s15 < 2^{14}, W$ }

MOVES.F Db, (EA)

MOVES.F Db, (SP+s15) { $-2^{14} \leq s15 < 2^{14}, W$ }

Description

This operation moves a fractional word from a data register to memory. The data is scaled according to the scaling mode and is saturated if the Ln bit is set. The address register values must be word-aligned. This instruction is affected by SM (Saturation Mode bit - SR[2]). When SM is set, scaling is not performed, and the scale bits S[1:10] have no effect. The address of the access must be word-aligned.

MOVES.F Db,(a16)

Writes the HP of a data register (Db) to an absolute 16-bit memory address.

MOVES.F Db,(a32)

Writes the HP of a data register (Db) to an absolute 32-bit memory address.

MOVES.F Db,(Rn+s15)

Writes the HP of a data register (Db) to a memory address pointed to by an address register (Rn) with a signed 15-bit offset.

MOVES.F Db,(EA)

Writes the HP of a data register (Db) to a memory address pointed to by an address register (EA) with an optional offset or post-increment.

MOVES.F Db,(SP+s15)

Writes the HP of a data register (Db) to a memory address pointed to by the active stack pointer (SP) with a signed 15-bit offset.

Status and Conditions that Affect Instruction

Register Address	Bit Name	Description
MCTL[31:0]	AM3–AM0	Address modification bits for R0–R7.
SR[5:4]	S[1:0]	Scaling mode bits choose: no scaling, scale up one bit, or scale down one bit.
Ln	L	Limited values are written to the destination if the Ln bit is set.
SR[18]	EXP	Determines the stack pointer used in instructions that have a stack pointer as an operand.

Status and Conditions Changed by Instruction

Register Address	Bit Name	Description
SR[6]	S	Scaling bit, set when the absolute value of the data moved (after scaling and limiting) is greater than or equal to 0.25 and less than 0.75.

Example

`moves.f d0, (r0)`

Register/Memory Address	Before	After
SR	\$00E0 0000	\$00E0 0000
R0	\$0000 0050	
L0:D0	\$1:\$00 8000 0000	
(\$0050)		\$7FFF

The Ln bit is set in d0, and the number in d0 is positive (bit 39 = 0), so the saturated value \$7FFF is written to memory.

Instruction Formats and Opcodes

Instruction	Words	Cycles	Type	Opcode																
MOVES.F Db, (a16)	2	1	3	<table border="1"> <tr> <td>15</td> <td>8</td> <td>7</td> <td>0</td> </tr> <tr> <td>0 0 0 0 0 j j j</td> <td>A A A 0 1 1 1 1</td> <td></td> <td></td> </tr> <tr> <td>1 0 0 A A A A A</td> <td>A A A A A A A A</td> <td></td> <td></td> </tr> </table>	15	8	7	0	0 0 0 0 0 j j j	A A A 0 1 1 1 1			1 0 0 A A A A A	A A A A A A A A						
15	8	7	0																	
0 0 0 0 0 j j j	A A A 0 1 1 1 1																			
1 0 0 A A A A A	A A A A A A A A																			
MOVES.F Db, (a32)	3	1	3	<table border="1"> <tr> <td>15</td> <td>8</td> <td>7</td> <td>0</td> </tr> <tr> <td>0 0 0 0 0 j j j</td> <td>A A A a a 0 1 1</td> <td></td> <td></td> </tr> <tr> <td>0 0 1 A A A A A</td> <td>A A A A A A A A</td> <td></td> <td></td> </tr> <tr> <td>1 0 a a a a a a</td> <td>a a a a a a a a</td> <td></td> <td></td> </tr> </table>	15	8	7	0	0 0 0 0 0 j j j	A A A a a 0 1 1			0 0 1 A A A A A	A A A A A A A A			1 0 a a a a a a	a a a a a a a a		
15	8	7	0																	
0 0 0 0 0 j j j	A A A a a 0 1 1																			
0 0 1 A A A A A	A A A A A A A A																			
1 0 a a a a a a	a a a a a a a a																			
MOVES.F Db, (Rn+s15)	2	2	3	<table border="1"> <tr> <td>15</td> <td>8</td> <td>7</td> <td>0</td> </tr> <tr> <td>0 0 0 0 0 j j j</td> <td>1 s s 1 0 R R R</td> <td></td> <td></td> </tr> <tr> <td>1 0 0 s s s s s</td> <td>s s s s s s s s</td> <td></td> <td></td> </tr> </table>	15	8	7	0	0 0 0 0 0 j j j	1 s s 1 0 R R R			1 0 0 s s s s s	s s s s s s s s						
15	8	7	0																	
0 0 0 0 0 j j j	1 s s 1 0 R R R																			
1 0 0 s s s s s	s s s s s s s s																			
MOVES.F Db, (EA)	1	1 ²	1	<table border="1"> <tr> <td>15</td> <td>8</td> <td>7</td> <td>0</td> </tr> <tr> <td>0 * 0 0 0 j j j</td> <td>0 1 M M M R R R</td> <td></td> <td></td> </tr> </table>	15	8	7	0	0 * 0 0 0 j j j	0 1 M M M R R R										
15	8	7	0																	
0 * 0 0 0 j j j	0 1 M M M R R R																			
Notes: <ol style="list-style-type: none"> ** indicates serial grouping encoding. When the form (Rn + N0) is used in EA, the cycle count is increased by 1. 																				
MOVES.F Db, (SP+s15)	2	2	3	<table border="1"> <tr> <td>15</td> <td>8</td> <td>7</td> <td>0</td> </tr> <tr> <td>0 0 0 0 0 j j j</td> <td>1 s s 1 1 1 1 0</td> <td></td> <td></td> </tr> <tr> <td>1 0 0 s s s s s</td> <td>s s s s s s s s</td> <td></td> <td></td> </tr> </table>	15	8	7	0	0 0 0 0 0 j j j	1 s s 1 1 1 1 0			1 0 0 s s s s s	s s s s s s s s						
15	8	7	0																	
0 0 0 0 0 j j j	1 s s 1 1 1 1 0																			
1 0 0 s s s s s	s s s s s s s s																			

Instruction Fields

Db **jjj** **Single Source/Destination Data Register**

000	D0	010	D2	100	D4	110	D6
001	D1	011	D3	101	D5	111	D7

Note: This instruction can specify D8-D15 as operands by using a prefix.

EA **MMM** **Effective Address Notation**

000	(Rn+N0)	010	(Rn)	100	(Rn)+N0	110	(Rn)+N2
001	(Rn)−	011	(Rn)+	101	(Rn)+N1	111	(Rn)+N3

Rn **RRR** **Address Register**

000	R0	010	R2	100	R4	110	R6
001	R1	011	R3	101	R5	111	R7

Note: This instruction can specify R8-R15 as operands by using a high register prefix.

a16 AAAAAAAAAAAAAAAAAA 16-bit unsigned absolute address

a32 aaaaaaaaaaaaaaaaaAAAAAAAAAAAAAAAAA 32-bit absolute long address



3.5

ssssssssssssss

Signed 15-bit offset



MOVES.L

Move Long to Memory With Scaling and Saturation (AGU)

MOVES.L

Operation

Db → (EA)

Assembler Syntax

MOVES.L Db, (EA)

Description

The data is scaled according to the scaling mode, and saturated if the Ln bit is set. The address register values must be long word-aligned. This instruction is affected by SM (Saturation Mode bit - SR[2]). When SM is set, scaling is not performed, and the scale bits S[1:10] have no effect

MOVES.L Db,(EA)

Moves a saturated long word from a data register (Db) to a memory address pointed to by an address register with an optional offset or post-increment.

Status and Conditions that Affect Instruction

Register Address	Bit Name	Description
MCTL[31:0]	AM3-AM0	Address modification bits when updating R0-R7. Otherwise, the instruction is not affected by MCTL.
SR[5:4]	S[1:0]	Scaling mode bits choose: no scaling, scale up one bit, or scale down one bit.
Ln	L	Limited values are written to the destination if the Ln bit is set.

Status and Conditions Changed by Instruction

Register Address	Bit Name	Description
SR[6]	S	Scaling bit, set when the absolute value of the data moved (after scaling and limiting) is greater than or equal to 0.25 and less than 0.75.

Example

```
moves.l d0, (r0)
```

Register/Memory Address	Before	After
SR	\$00E0 0000	\$00E0 0000
R0	\$0000 0054	
L0:D0	\$1:\$00 8000 0000	
\$00000054		\$7FFF FFFF

Instruction Formats and Opcodes

Instruction	Words	Cycles	Type	Opcode																
MOVES.L Db, (EA)	1	1 ²	1	<div style="display: flex; justify-content: space-between; align-items: center;"> 15 8 7 0 </div> <table border="1" style="width: 100%; text-align: center;"> <tr> <td style="width: 15%;">0</td> <td style="width: 15%;">*</td> <td style="width: 15%;">0</td> <td style="width: 15%;">0</td> <td style="width: 15%;">0</td> <td style="width: 15%;">j</td> <td style="width: 15%;">j</td> <td style="width: 15%;">j</td> <td style="width: 15%;">1</td> <td style="width: 15%;">1</td> <td style="width: 15%;">M</td> <td style="width: 15%;">M</td> <td style="width: 15%;">M</td> <td style="width: 15%;">R</td> <td style="width: 15%;">R</td> <td style="width: 15%;">R</td> </tr> </table>	0	*	0	0	0	j	j	j	1	1	M	M	M	R	R	R
0	*	0	0	0	j	j	j	1	1	M	M	M	R	R	R					

- Notes:**
1. ** indicates serial grouping encoding.
 2. When the form (Rn + N0) is used in EA, the cycle count is increased by 1.

Instruction Fields

Rn	RRR	Address Register						
	000	R0	010	R2	100	R4	110	R6
	001	R1	011	R3	101	R5	111	R7

Note: This instruction can specify R8-R15 as operands by using a high register prefix.

Db	jjj	Single Source/Destination Data Register						
	000	D0	010	D2	100	D4	110	D6
	001	D1	011	D3	101	D5	111	D7

Note: This instruction can specify D8-D15 as operands by using a prefix.

EA	MMM	Effective Address Notation						
	000	(Rn+N0)	010	(Rn)	100	(Rn)+N0	110	(Rn)+N2
	001	(Rn)–	011	(Rn)+	101	(Rn)+N1	111	(Rn)+N3



MOVEU.B

Move Unsigned Byte from Memory (AGU)

MOVEU.B

Operation

(aa) → DR

(aa) → DR

(Rn + s15) → DR

(ea) → DR

(SP + s15) → DR

Assembler Syntax

MOVEU.B (a16),DR { $0 \leq a16 < 2^{16}$ }

MOVEU.B (a32),DR { $0 \leq a32 < 2^{32}$ }

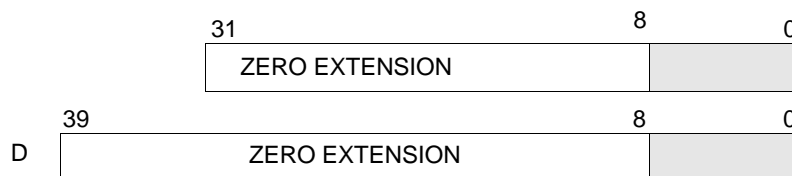
MOVEU.B (Rn+s15),DR { $-2^{14} \leq s15 < 2^{14}$ }

MOVEU.B (ea),DR

MOVEU.B (SP+s15),DR { $-2^{14} \leq s15 < 2^{14}$ }

Description

These operations move an unsigned byte from memory into a data or address register (DR). Data is placed in bits 7:0 of the destination register (DR) and zero-extended.



MOVEU.B (a16),DR

Reads an unsigned byte from a 16-bit unsigned absolute address in memory into a data or address register (DR).

MOVEU.B (a32),DR

Reads an unsigned byte from an absolute 32-bit address in memory into a data or address register (DR).

MOVEU.B (Rn+s15),DR

Reads an unsigned byte from a memory address pointed to by an address register with a signed 15-bit offset into a data or address register (DR).

MOVEU.B (ea),DR

Reads an unsigned byte from a memory address pointed to by an address register with an optional offset or post-increment into a data or address register (DR).

MOVEU.B (SP+s15),DR

Reads an unsigned byte from a memory address pointed to by the active stack pointer with a signed 15-bit offset into a data or address register (DR).



Status and Conditions that Affect Instruction

Register Address	Bit Name	Description
MCTL[31:0]	AM3–AM0	Address modification bits when updating R0–R7. Otherwise, the instruction is not affected by MCTL.
SR[18]	EXP	Determines which stack pointer is used when the stack pointer is an operand. Otherwise, the instruction is not affected by SR.

Status and Conditions Changed by Instruction

Register Address	Bit Name	Description
Ln	L	Clears the Ln bit in the destination registers.

Example

```
moveu.b ($0053),d10
```

Register/Memory Address	Before	After
(\$0053)	\$F8	
D10		\$0:\$00 0000 00F8



Instruction Formats and Opcodes

Instruction	Words	Cycles	Type	Opcode																
MOVEU.B (a16),DR	2	1	3	<table border="1"> <tr> <td>15</td> <td>8</td> <td>7</td> <td>0</td> </tr> <tr> <td>0 0 0 1 H H H H</td> <td>A A A 0 1 1 0 0</td> <td></td> <td></td> </tr> <tr> <td>1 0 0 A A A A A</td> <td>A A A A A A A A</td> <td></td> <td></td> </tr> </table>	15	8	7	0	0 0 0 1 H H H H	A A A 0 1 1 0 0			1 0 0 A A A A A	A A A A A A A A						
15	8	7	0																	
0 0 0 1 H H H H	A A A 0 1 1 0 0																			
1 0 0 A A A A A	A A A A A A A A																			
MOVEU.B (a32),DR	3	1	3	<table border="1"> <tr> <td>15</td> <td>8</td> <td>7</td> <td>0</td> </tr> <tr> <td>0 0 0 0 H H H H</td> <td>A A A a a 1 0 1</td> <td></td> <td></td> </tr> <tr> <td>0 0 1 A A A A A</td> <td>A A A A A A A A</td> <td></td> <td></td> </tr> <tr> <td>1 0 a a a a a a</td> <td>a a a a a a a a</td> <td></td> <td></td> </tr> </table>	15	8	7	0	0 0 0 0 H H H H	A A A a a 1 0 1			0 0 1 A A A A A	A A A A A A A A			1 0 a a a a a a	a a a a a a a a		
15	8	7	0																	
0 0 0 0 H H H H	A A A a a 1 0 1																			
0 0 1 A A A A A	A A A A A A A A																			
1 0 a a a a a a	a a a a a a a a																			
MOVEU.B (Rn+s15),DR	2	2	3	<table border="1"> <tr> <td>15</td> <td>8</td> <td>7</td> <td>0</td> </tr> <tr> <td>0 0 0 1 H H H H</td> <td>0 s s 1 0 R R R</td> <td></td> <td></td> </tr> <tr> <td>1 0 0 s s s s s</td> <td>s s s s s s s s</td> <td></td> <td></td> </tr> </table>	15	8	7	0	0 0 0 1 H H H H	0 s s 1 0 R R R			1 0 0 s s s s s	s s s s s s s s						
15	8	7	0																	
0 0 0 1 H H H H	0 s s 1 0 R R R																			
1 0 0 s s s s s	s s s s s s s s																			
MOVEU.B (ea),DR	1	1 ¹	4	<table border="1"> <tr> <td>15</td> <td>8</td> <td>7</td> <td>0</td> </tr> <tr> <td>1 0 0 1 H H H H</td> <td>1 0 1 M M R R R</td> <td></td> <td></td> </tr> </table>	15	8	7	0	1 0 0 1 H H H H	1 0 1 M M R R R										
15	8	7	0																	
1 0 0 1 H H H H	1 0 1 M M R R R																			
MOVEU.B (SP+s15),DR	2	2	3	<table border="1"> <tr> <td>15</td> <td>8</td> <td>7</td> <td>0</td> </tr> <tr> <td>0 0 0 1 H H H H</td> <td>0 s s 1 1 1 0 0</td> <td></td> <td></td> </tr> <tr> <td>1 0 0 s s s s s</td> <td>s s s s s s s s</td> <td></td> <td></td> </tr> </table>	15	8	7	0	0 0 0 1 H H H H	0 s s 1 1 1 0 0			1 0 0 s s s s s	s s s s s s s s						
15	8	7	0																	
0 0 0 1 H H H H	0 s s 1 1 1 0 0																			
1 0 0 s s s s s	s s s s s s s s																			

Note 1: When the form (Rn + N0) is used in ea, the cycle count is increased by 1.

Instruction Fields

DR

HHHH

Data/Address Register

0000	D0	0100	D4	1000	R0	1100	R4
0001	D1	0101	D5	1001	R1	1101	R5
0010	D2	0110	D6	1010	R2	1110	R6
0011	D3	0111	D7	1011	R3	1111	R7

Note: This instruction can specify D8-D15 or R8-R15 as operands by using a high register prefix.

ea

MM

Effective Address Notation

00	(Rn)+	01	(Rn)-	10	(Rn+N0)	11	(Rn)
----	-------	----	-------	----	---------	----	------

Rn

RRR

Address Register

000	R0	010	R2	100	R4	110	R6
001	R1	011	R3	101	R5	111	R7

Note: This instruction can specify R8-R15 as operands by using a high register prefix.

a16

AAAAAAAAAAAAAAAAAAAA 16-bit unsigned absolute address



a2

aaaaaaaaaaaaaaaaAAAAAAAAAAAAAAAA

32-bit absolute long address

s15

ssssssssssssss

Signed 15-bit offset



MOVEU.L

Move Unsigned Immediate Long to a Data Register (AGU)

MOVEU.L

Operation

#u32 → Db

Assembler Syntax

MOVEU.L #u32,Db {0 ≤ u32 < 2³²}

Description

MOVEU.L #u32,Db

Loads an unsigned long word (32-bit) immediate value into a data register (Db), zero-extending it.



Status and Conditions that Affect Instruction

None.

Status and Conditions Changed by Instruction

Register Address	Bit Name	Description
Ln	L	Clears the Ln bit in the destination register.

Example

```
moveu.l #$fffffff8,d3
```

Register/Memory Address	Before	After
Immediate	\$FFFF FFF8	
D3		\$0:\$00 FFFF FFFF8



Instruction Formats and Opcodes

Instruction	Words	Cycles	Type	Opcode						
MOVEU.L #u32,Db	3	1	3	<div style="display: flex; justify-content: space-between; width: 100%;"> 15 8 7 0 </div> <table border="1" style="width: 100%; border-collapse: collapse;"> <tr> <td style="width: 33.33%;">0 0 1 1 0 j j j</td> <td style="width: 33.33%;">i i i l l 0 0 1</td> </tr> <tr> <td>0 0 1 i i i i i</td> <td>i i i i i i i i</td> </tr> <tr> <td>1 0 l l l l l l</td> <td>l l l l l l l l</td> </tr> </table>	0 0 1 1 0 j j j	i i i l l 0 0 1	0 0 1 i i i i i	i i i i i i i i	1 0 l l l l l l	l l l l l l l l
0 0 1 1 0 j j j	i i i l l 0 0 1									
0 0 1 i i i i i	i i i i i i i i									
1 0 l l l l l l	l l l l l l l l									

Instruction Fields

Db	jjj	Single Source/Destination Data Register					
000	D0	010	D2	100	D4	110	D6
001	D1	011	D3	101	D5	111	D7

Note: This instruction can specify D8-D15 as operands by using a prefix.

#u32 (31)IIIIIIIIIIIIIIIIII(16) 32-bit unsigned immediate data
 (15)iiiiiiiiiiiiiiiiii (0)

MOVEU.W Move Unsigned Immediate Word to a Register Portion (AGU) MOVEU.W

Operation

#u16 → Db[31:16]

#u16 → Db[15:0]

Assembler Syntax

MOVEU.W #u16,Db.H {0 ≤ u16 < 2¹⁶}

MOVEU.W #u16,Db.L {0 ≤ u16 < 2¹⁶}

Description

These operations move an immediate unsigned word to a high/low part of a data register (Db) without changing the other bits in the data register (Db).

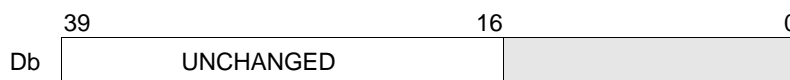
MOVEU.W #u16,Db.H

Loads an immediate unsigned word into the HP of a data register (Db). The other bits in the register are unchanged.



MOVEU.W #u16,Db.L

Loads an immediate unsigned word into the LP of a data register (Db). The other bits in the register are unchanged.



Status and Conditions that Affect Instruction

None.

Status and Conditions Changed by Instruction

Register Address	Bit Name	Description
Ln	L	Clears the Ln bit in the destination register.

Example

```
moveu.w #$2345,d10.l
```

Register/Memory Address	Before	After
Immediate	\$2345	
D10	\$0:\$00 ABCD EFFF	\$0:\$00 ABCD 2345



Instruction Formats and Opcodes

Instruction	Words	Cycles	Type	Opcode												
MOVEU.W #u16,Db.H	2	1	3	<table border="1"> <tr> <td style="text-align: right;">15</td> <td style="text-align: right;">8</td> <td style="text-align: right;">7</td> <td style="text-align: right;">0</td> </tr> <tr> <td>0 0 0 1 1 0 0 1</td> <td>i i i 1 0 j j j</td> <td></td> <td></td> </tr> <tr> <td>1 0 1 i i i i i</td> <td>i i i i i i i i</td> <td></td> <td></td> </tr> </table>	15	8	7	0	0 0 0 1 1 0 0 1	i i i 1 0 j j j			1 0 1 i i i i i	i i i i i i i i		
15	8	7	0													
0 0 0 1 1 0 0 1	i i i 1 0 j j j															
1 0 1 i i i i i	i i i i i i i i															
MOVEU.W #u16,Db.L	2	1	3	<table border="1"> <tr> <td style="text-align: right;">15</td> <td style="text-align: right;">8</td> <td style="text-align: right;">7</td> <td style="text-align: right;">0</td> </tr> <tr> <td>0 0 0 1 1 0 0 1</td> <td>i i i 0 0 j j j</td> <td></td> <td></td> </tr> <tr> <td>1 0 1 i i i i i</td> <td>i i i i i i i i</td> <td></td> <td></td> </tr> </table>	15	8	7	0	0 0 0 1 1 0 0 1	i i i 0 0 j j j			1 0 1 i i i i i	i i i i i i i i		
15	8	7	0													
0 0 0 1 1 0 0 1	i i i 0 0 j j j															
1 0 1 i i i i i	i i i i i i i i															

Instruction Fields

Db	jjj	Single Source/Destination Data Register					
		000	D0	010	D2	100	D4
001	D1	011	D3	101	D5	111	D7

Note: This instruction can specify D8-D15 as operands by using a prefix.

#u16 iiiiiiiiiiiiiii 16-bit unsigned immediate data



MOVEU.W

Move Unsigned Word from Memory to a Register (AGU)

MOVEU.W

Operation

(aa) → C4

(aa) → DR

(Rn + s15) → DR

(EA) → DR

(SP + s15) → C4

Assembler Syntax

MOVEU.W (a16),C4 { $0 \leq a16 < 2^{16}$ }

MOVEU.W (a32),DR { $0 \leq a32 < 2^{32}$ }

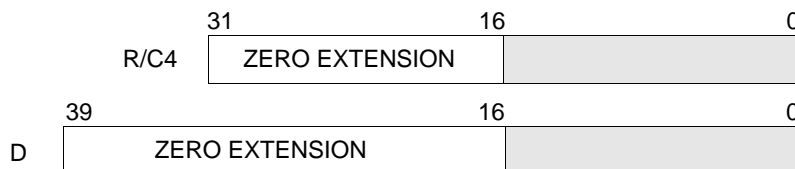
MOVEU.W (Rn+s15),DR { $-2^{14} \leq s15 < 2^{14}$ }

MOVEU.W (EA),DR

MOVEU.W (SP+s15),C4 { $-2^{14} \leq s15 < 2^{14}$ }

Description

These operations move an unsigned word from memory to the LP of a register and zero-extend it. The address of the access must be word-aligned.



MOVEU.W (a16),C4

Reads an unsigned word from an 16-bit unsigned absolute address, places the data in the LP of a general register (C4), and zero-extends the upper bits.

MOVEU.W (a32),DR

Reads an unsigned word from an absolute 32-bit address, places the data in the LP of a data or address register (DR), and zero-extends the upper bits.

MOVEU.W (Rn+s15),DR

Reads an unsigned word from a memory address pointed to by an address register (Rn) with a signed 15-bit offset, places the data in the LP of a data or address register (DR), and zero-extends the upper bits.

MOVEU.W (EA),DR

Reads an unsigned word from a memory address pointed to by an address register with an optional offset or post-increment, places the data in the LP of a data or address register (DR), and zero-extends the upper bits.

MOVEU.W (SP+s15),C4

Reads an unsigned word from a memory address pointed to by the active stack pointer (SP) with a signed 15-bit offset, places the data in the LP of a general register (C4), and zero-extends the upper bits.

Status and Conditions that Affect Instruction

Register Address	Bit Name	Description
MCTL[31:0]	AM3–AM0	Address modification bits when updating R0–R7. Otherwise, the instruction is not affected by MCTL.
SR[18]	EXP	Determines which stack pointer is used when the stack pointer is an operand. Otherwise, the instruction is not affected by SR.

Status and Conditions Changed by Instruction

Register Address	Bit Name	Description
Ln	L	Clears the Ln bit in the destination registers.

Example

```
moveu.w (r7+2),d10
```

Register/Memory Address	Before	After
MCTL	\$0000 0000	
R7	\$0000 0050	
(R7+2)	\$FFF8	
D10	\$00 1010 0000	\$00 0000 FFF8

Instruction Formats and Opcodes

Instructions	Words	Cycles	Type	Opcode																
MOVEU.W (a16),C4	2	1	3	<table border="1"> <tr> <td>15</td> <td>8</td> <td>7</td> <td>0</td> </tr> <tr> <td>0 0 0 1 D D D D</td> <td>A A A 0 1 1 D 1</td> <td></td> <td></td> </tr> <tr> <td>1 0 0 A A A A A</td> <td>A A A A A A A A</td> <td></td> <td></td> </tr> </table>	15	8	7	0	0 0 0 1 D D D D	A A A 0 1 1 D 1			1 0 0 A A A A A	A A A A A A A A						
15	8	7	0																	
0 0 0 1 D D D D	A A A 0 1 1 D 1																			
1 0 0 A A A A A	A A A A A A A A																			
MOVEU.W (a32),DR	3	1	3	<table border="1"> <tr> <td>15</td> <td>8</td> <td>7</td> <td>0</td> </tr> <tr> <td>0 0 0 0 H H H H</td> <td>A A A a a 1 1 1</td> <td></td> <td></td> </tr> <tr> <td>0 0 1 A A A A A</td> <td>A A A A A A A A</td> <td></td> <td></td> </tr> <tr> <td>1 0 a a a a a a</td> <td>a a a a a a a a</td> <td></td> <td></td> </tr> </table>	15	8	7	0	0 0 0 0 H H H H	A A A a a 1 1 1			0 0 1 A A A A A	A A A A A A A A			1 0 a a a a a a	a a a a a a a a		
15	8	7	0																	
0 0 0 0 H H H H	A A A a a 1 1 1																			
0 0 1 A A A A A	A A A A A A A A																			
1 0 a a a a a a	a a a a a a a a																			
MOVEU.W (Rn+s15),DR	2	2	3	<table border="1"> <tr> <td>15</td> <td>8</td> <td>7</td> <td>0</td> </tr> <tr> <td>0 0 0 1 H H H H</td> <td>1 s s 1 0 R R R</td> <td></td> <td></td> </tr> <tr> <td>1 0 0 s s s s s</td> <td>s s s s s s s s</td> <td></td> <td></td> </tr> </table>	15	8	7	0	0 0 0 1 H H H H	1 s s 1 0 R R R			1 0 0 s s s s s	s s s s s s s s						
15	8	7	0																	
0 0 0 1 H H H H	1 s s 1 0 R R R																			
1 0 0 s s s s s	s s s s s s s s																			
MOVEU.W (EA),DR	1	1 ²	1	<table border="1"> <tr> <td>15</td> <td>8</td> <td>7</td> <td>0</td> </tr> <tr> <td>0 * 0 1 H H H H</td> <td>1 1 M M M R R R</td> <td></td> <td></td> </tr> </table>	15	8	7	0	0 * 0 1 H H H H	1 1 M M M R R R										
15	8	7	0																	
0 * 0 1 H H H H	1 1 M M M R R R																			
Notes: <ol style="list-style-type: none"> ** indicates serial grouping encoding. When the form (Rn + N0) is used in EA, the cycle count is increased by 1. 																				
MOVEU.W (SP+s15),C4	2	2	3	<table border="1"> <tr> <td>15</td> <td>8</td> <td>7</td> <td>0</td> </tr> <tr> <td>0 0 0 1 D D D D</td> <td>1 s s 1 1 1 D 0</td> <td></td> <td></td> </tr> <tr> <td>1 0 0 s s s s s</td> <td>s s s s s s s s</td> <td></td> <td></td> </tr> </table>	15	8	7	0	0 0 0 1 D D D D	1 s s 1 1 1 D 0			1 0 0 s s s s s	s s s s s s s s						
15	8	7	0																	
0 0 0 1 D D D D	1 s s 1 1 1 D 0																			
1 0 0 s s s s s	s s s s s s s s																			

Instruction Fields

C4	DDDDD	General Registers					
00000	D0	01000	D4	10000	R0	11000	R4
00001	B0	01001	B4	10001	N0	11001	M0
00010	D1	01010	D5	10010	R1	11010	R5
00011	B1	01011	B5	10011	N1	11011	M1
00100	D2	01100	D6	10100	R2	11100	R6
00101	B2	01101	B6	10101	N2	11101	M2
00110	D3	01110	D7	10110	R3	11110	R7
00111	B3	01111	B7	10111	N3	11111	M3

Note: This instruction can specify D8-D15 or R8-R15 as operands by using a high register prefix.

DR	HHHH	Data/Address Register					
0000	D0	0100	D4	1000	R0	1100	R4
0001	D1	0101	D5	1001	R1	1101	R5
0010	D2	0110	D6	1010	R2	1110	R6
0011	D3	0111	D7	1011	R3	1111	R7

Note: This instruction can specify D8-D15 or R8-R15 as operands by using a high register prefix.



Operation

Da.H * Db.H → Dn

Assembler Syntax

MPY Da, Db, Dn

Description

MPY Da,Db,Dn

Performs signed fractional multiplication of the high portions of two data registers (Da, Db) and stores the product in a destination data register (Dn).

Status and Conditions that Affect Instruction

Register Address	Bit Name	Description
SR[2]	SM	If set, selects 32-bit arithmetic saturation mode.
SR[5:4]	S[1:0]	Scaling bits determine which bits in the result are used in the Ln bit calculation.

Status and Conditions Changed by Instruction

Register Address	Bit Name	Description
Ln	L	If not in arithmetic saturation mode (SR [SM] = 0), calculates and updates the Ln bit in the destination register. If in arithmetic saturation mode (SR [SM] = 1), clears the Ln bit in the destination register.
EMR[2]	DOVF	Set if the result cannot be represented in 40 bits, or if the result saturates to 32 bits in saturation mode.

Example 1

mpy d4, d5, d6

Register/Memory Address	Before	After
SR	\$00E0 0000	
D4	\$FF C000 0000	
D5	\$00 2000 0000	
L6:D6		\$0:\$FF F000 0000
EMR		\$0000 0000

$$\begin{array}{r}
 0.010 \quad \$2000 \quad 1/4 \\
 \times 1.100 \quad \$C000 \quad -1/2 \\
 \hline
 1.111 \quad \$F000 \quad -1/8
 \end{array}$$

Example 2

mpy d6 ,d6 ,d7

Register/Memory Address	Before	After
D6	\$FF C000 0000	
L7:D7		\$0:\$00 2000 0000

Instruction Formats and Opcodes

Instruction	Words	Cycles	Type	Opcode
MPY Da , Da , Dn	1	1	1	<div style="display: flex; justify-content: space-between; width: 100%;"> 15 8 7 0 </div> <div style="border: 1px solid black; padding: 2px; display: flex; justify-content: space-between;"> 0 * 1 0 1 0 F F F 1 1 1 0 0 j j </div>
MPY Da , Db , Dn	1	1	1	<div style="display: flex; justify-content: space-between; width: 100%;"> 15 8 7 0 </div> <div style="border: 1px solid black; padding: 2px; display: flex; justify-content: space-between;"> 0 * 1 0 0 0 F F F 0 1 J J J J J </div>

Note: ** indicates serial grouping encoding.

Instruction Fields

Da, Da	jj	Data Register Pairs							
		00	D1,D1	01	D3,D3	10	D5,D5	11	D7,D7

Note: This instruction can specify D8-D15 as operands by using a prefix.

Da, Db	JJJJ	Data Register Pairs						
	0000	D0,D4	01000	D2,D4	10000	D0,D0	11000	D1,D2
	00001	D0,D5	01001	D2,D5	10001	D0,D1	11001	D1,D3
	00010	D0,D6	01010	D2,D6	10010	D0,D2	11010	D5,D6
	00011	D0,D7	01011	D2,D7	10011	D0,D3	11011	D5,D7
	00100	D1,D4	01100	D3,D4	10100	D4,D4	11100	D2,D2
	00101	D1,D5	01101	D3,D5	10101	D4,D5	11101	D2,D3
	00110	D1,D6	01110	D3,D6	10110	D4,D6	11110	D6,D6
	00111	D1,D7	01111	D3,D7	10111	D4,D7	11111	D6,D7

- Notes:**
1. This instruction can specify D8-D15 as operands by using a prefix.
 2. Register pair order can be inverted for clarity because the order of operation is not important for multiply operations.
 3. The JJJJ encoding does not include the pairs: D1–D1, D3–D3, D5–D5, and D7–D7. These are covered in the jj encoding.

FFF **Single Source/Destination Data Register**

000	D0	010	D2	100	D4	110	D6
001	D1	011	D3	101	D5	111	D7

Note: This instruction can specify D8-D15 as operands by using a prefix.

and Round (DALU)

Operation

Rnd((Da.H * Db.H)) → Dn

Assembler Syntax

MPYR Da, Db, Dn

Description

MPYR Da,Db,Dn

Performs signed fractional multiplication of the high portions of a data register pair (Da, Db), rounds the product, and stores the result in a destination data register (Dn). Rounding adjusts the LSB of the high part of the destination register according to the value of the low part of the register and then zeros the low part. The two modes of the round function, Rnd(), are described on page A-359.

Status and Conditions that Affect Instruction

Register Address	Bit Name	Description
SR[2]	SM	If set, selects 32-bit arithmetic saturation mode.
SR[3]	RM	Rounding mode
SR[5:4]	S[1:0]	Scaling bits determine which bits in the result are used in the Ln bit calculation and which bits are used in rounding.

Status and Conditions Changed by Instruction

Register Address	Bit Name	Description
Ln	L	If not in arithmetic saturation mode (SR [SM] = 0), calculates and updates the Ln bit in the destination register. If in arithmetic saturation mode (SR [SM] = 1), clears the Ln bit in the destination register.
EMR[2]	DOVF	Set if the result cannot be represented in 40 bits, or if the result saturates to 32 bits in saturation mode.

Example

mpyr d4, d5, d6

Register/Memory Address	Before	After
SR	\$00E0 0000	
D4	\$00 4001 0000	
D5	\$00 4002 0000	

Register/Memory Address	Before	After
L6:D6		\$0:\$00 2002 0000
EMR		\$0000 0000

```

0.100 0000 0000 0001$4001
x 0.100 0000 0000 0010$4002
-----
0.010 0000 0000 0001 1000 0000 0000 0000$2001 8000
rounded 0.010 0000 0000 0010$2002

```

Instruction Formats and Opcodes

Instruction	Words	Cycles	Type	Opcode									
MPYR Da, Db, Dn	1	1	1	<table border="1"> <tr> <td>15</td> <td>8</td> <td>7</td> <td>0</td> </tr> <tr> <td>0</td> <td>*</td> <td>1</td> <td>0 0 1 F F</td> <td>F 0 1 J J J J J</td> </tr> </table>	15	8	7	0	0	*	1	0 0 1 F F	F 0 1 J J J J J
15	8	7	0										
0	*	1	0 0 1 F F	F 0 1 J J J J J									
MPYR Da, Da, Dn	1	1	1	<table border="1"> <tr> <td>15</td> <td>8</td> <td>7</td> <td>0</td> </tr> <tr> <td>0</td> <td>*</td> <td>1</td> <td>0 0 0 F F</td> <td>F 1 1 0 1 0 j j</td> </tr> </table>	15	8	7	0	0	*	1	0 0 0 F F	F 1 1 0 1 0 j j
15	8	7	0										
0	*	1	0 0 0 F F	F 1 1 0 1 0 j j									

Note: ** indicates serial grouping encoding.

Instruction Fields

Da,Db	JJJJ	Data Register Pairs					
00000	D0,D4	01000	D2,D4	10000	D0,D0	11000	D1,D2
00001	D0,D5	01001	D2,D5	10001	D0,D1	11001	D1,D3
00010	D0,D6	01010	D2,D6	10010	D0,D2	11010	D5,D6
00011	D0,D7	01011	D2,D7	10011	D0,D3	11011	D5,D7
00100	D1,D4	01100	D3,D4	10100	D4,D4	11100	D2,D2
00101	D1,D5	01101	D3,D5	10101	D4,D5	11101	D2,D3
00110	D1,D6	01110	D3,D6	10110	D4,D6	11110	D6,D6
00111	D1,D7	01111	D3,D7	10111	D4,D7	11111	D6,D7

- Notes:**
1. This instruction can specify D8-D15 as operands by using a prefix.
 2. Register pair order can be inverted for clarity because the order of operation is not important for multiply operations.
 3. The JJJJ encoding does not include the pairs: D1-D1, D3-D3, D5-D5, and D7-D7. These are covered in the jj encoding.

Da,Da	jj	Data Register Pairs					
00	D1,D1	01	D3,D3	10	D5,D5	11	D7,D7

Note: This instruction can specify D8-D15 as operands by using a prefix.

Dn	FFF	Single Source/Destination Data Register					
000	D0	010	D2	100	D4	110	D6
001	D1	011	D3	101	D5	111	D7

Note: This instruction can specify D8-D15 as operands by using a prefix.





Operation

$Dc.H * Dd.L \rightarrow Dn$

Assembler Syntax

MPYSU *Dc, Dd, Dn*

Description

MPYSU *Dc,Dd,Dn*

Performs signed fractional multiplication between the signed 16-bit HP of the first register (*Dc*) of a data register pair with the unsigned 16-bit LP of the second register (*Dd*). It then stores the sign-extended 32-bit product in a destination data register (*Dn*).

Status and Conditions that Affect Instruction

None.

Status and Conditions Changed by Instruction

Register Address	Bit Name	Description
<i>Ln</i>	L	Clears the <i>Ln</i> bit in the destination registers.

Example

mpysu *d4, d5, d6*

Register/Memory Address	Before	After
D4	\$FF C000 0001	
D5	\$FF E000 0002	
L6:D6		\$0:\$FF FFFF 0000

Instruction Formats and Opcodes

Instruction	Words	Cycles	Type	Opcode
MPYSU <i>Dc, Dd, Dn</i>	1	1	1	<div style="display: flex; justify-content: space-between; align-items: center;"> 15 8 7 0 </div> <div style="border: 1px solid black; padding: 2px; display: flex; align-items: center;"> 0 * 1 0 1 1 F F F 1 1 0 1 0 e e </div>

Note: ** indicates serial grouping encoding.

Instruction Fields

Dc,Dd	ee	Data Register Pairs					
00	D0,D1	01	D2,D3	10	D4,D5	11	D6,D7

Note: This instruction can specify D8-D15 as operands by using a prefix.



FFF Single Source/Destination Data Register

000	D0	010	D2	100	D4	110	D6
001	D1	011	D3	101	D5	111	D7

Note: This instruction can specify D8-D15 as operands by using a prefix.



Operation

$Dc.L * Dd.H \rightarrow Dn$

Assembler Syntax

MPYUS *Dc, Dd, Dn*

Description

MPYUS *Dc,Dd,Dn*

Performs signed fractional multiplication between the unsigned 16-bit LP of the first register (*Dc*) of a data register pair with the signed 16-bit HP of the second register (*Dd*). It then stores the sign-extended 32-bit product in a destination data register (*Dn*).

Status and Conditions that Affect Instruction

None.

Status and Conditions Changed by Instruction

Register Address	Bit Name	Description
<i>Ln</i>	<i>L</i>	Clears the <i>Ln</i> bit in the destination registers.

Example

`mpyus d2,d3,d4`

Register/Memory Address	Before	After
D2	\$FF FF00 0002	
D3	\$FF C000 0042	
L4:D4		\$0:\$FF FFFF 0000

```

1.100 $C000 (-2-1)
x 0.000 0000 0000 0010$0002 (2-14)
1.111 1111 1111 1111$FFFF (-2-15)

```

Instruction Formats and Opcodes

Instruction	Words	Cycles	Type	Opcode
MPYUS <i>Dc, Dd, Dn</i>	1	1	1	<div style="display: flex; justify-content: space-between; align-items: center;"> 15 8 7 0 </div> <div style="border: 1px solid black; padding: 2px; display: flex; justify-content: space-between; align-items: center;"> 0 * 1 0 0 0 F F F 1 1 1 0 1 e e </div>

Note: ** indicates serial grouping encoding.



struction Fields

Dc,Dd

ee

Data Register Pairs

00	D0,D1	01	D2,D3	10	D4,D5	11	D6,D7
----	-------	----	-------	----	-------	----	-------

Note: This instruction can specify D8-D15 as operands by using a prefix.

Dn

FFF

Single Source/Destination Data Register

000	D0	010	D2	100	D4	110	D6
001	D1	011	D3	101	D5	111	D7

Note: This instruction can specify D8-D15 as operands by using a prefix.



MPYUU

Fractional Multiply Unsigned By Unsigned (DALU)

MPYUU

Operation

$$Dc.L * Dd.L \rightarrow Dn$$

Assembler Syntax

$$MPYUU \ Dc, Dd, Dn$$

Description

MPYUU Dc,Dd,Dn

Performs unsigned fractional multiplication between the unsigned 16-bit LP of the first register (Dc) of a data register pair with the unsigned 16-bit LP of the second register (Dd). It then stores the sign-extended 32-bit product in a destination data register (Dn).

Status and Conditions that Affect Instruction

None.

Status and Conditions Changed by Instruction

Register Address	Bit Name	Description
Ln	L	Clears the Ln bit in the destination registers.

Example

```
mpyuu d4, d5, d6
```

Register/Memory Address	Before	After
D4	\$00 4000 2000	
D5	\$FF E000 4000	
L6:D6		\$0:\$00 1000 0000

$$\begin{array}{r} 0.010 \ \$2000 \ (2^{-2}) \\ \times 0.100\$4000 \ (2^{-1}) \\ \hline 0.001 \ \$1000 \ (2^{-3}) \end{array}$$

Instruction Formats and Opcodes

Instruction	Words	Cycles	Type	Opcode
MPYUU Dc, Dd, Dn	1	1	1	<div style="display: flex; justify-content: space-between; align-items: center;"> 15 8 7 0 </div> <div style="border: 1px solid black; padding: 2px; display: flex; justify-content: space-between; align-items: center;"> 0 * 1 0 1 1 F F F 1 1 0 1 1 e e </div>

Note: ** indicates serial grouping encoding.

struction Fields

Dc,Dd	ee	Data Register Pairs					
		00	D0,D1	01	D2,D3	10	D4,D5

Note: This instruction can specify D8-D15 as operands by using a prefix.

Dn	FFF	Single Source/Destination Data Register							
		000	D0	010	D2	100	D4	110	D6
		001	D1	011	D3	101	D5	111	D7

Note: This instruction can specify D8-D15 as operands by using a prefix.

Operation

0 – Dn → Dn

Assembler Syntax

NEG Dn

Description

NEG Dn

Negates the contents of a source data register (Dn) and stores the 40-bit two's complement result in a destination data register (Dn).

Status and Conditions that Affect Instruction

Register Address	Bit Name	Description
SR[2]	SM	If set, selects 32-bit arithmetic saturation mode.
SR[5:4]	S[1:0]	Scaling mode bits determine which bits in the result are used in the Ln bit calculation.

Status and Conditions Changed by Instruction

Register Address	Bit Name	Description
Ln	L	If not in arithmetic saturation mode (SR [SM] = 0), calculates and updates the Ln bit in the destination register. If in arithmetic saturation mode (SR [SM] = 1), clears the Ln bit in the destination register.
EMR[2]	DOVF	Set if the result cannot be represented in 40 bits, or if the result saturates to 32 bits in saturation mode.

Example

neg d3

Register/Memory Address	Before	After
SR	\$00E0 0000	
L3:D3	\$0:\$00 0765 1235	\$0:\$FF F89A EDCB
EMR		\$0000 0000

```

0000 0111 0110 0101 0001 0010 0011 0101
invert 1111 1000 1001 1010 1110 1101 1100 1010
add one 1111 1000 1001 1010 1110 1101 1100 1011

```

Instruction Formats and Opcodes

Instruction	Words	Cycles	Type	Opcode																
NEG Dn	1	1	1	<div style="display: flex; justify-content: space-between; align-items: center;"> 15 8 7 0 </div> <table border="1" style="width: 100%; text-align: center;"> <tr> <td style="width: 10%;">0</td> <td style="width: 10%;">*</td> <td style="width: 10%;">1</td> <td style="width: 10%;">0</td> <td style="width: 10%;">0</td> <td style="width: 10%;">1</td> <td style="width: 10%;">F</td> <td style="width: 10%;">F</td> <td style="width: 10%;">F</td> <td style="width: 10%;">1</td> <td style="width: 10%;">1</td> <td style="width: 10%;">0</td> <td style="width: 10%;">0</td> <td style="width: 10%;">1</td> <td style="width: 10%;">0</td> <td style="width: 10%;">0</td> </tr> </table>	0	*	1	0	0	1	F	F	F	1	1	0	0	1	0	0
0	*	1	0	0	1	F	F	F	1	1	0	0	1	0	0					

Note: ** indicates serial grouping encoding.

Instruction Fields

Dn	FFF	Single Source/Destination Data Register						
	000	D0	010	D2	100	D4	110	D6
	001	D1	011	D3	101	D5	111	D7

Note: This instruction can specify D8-D15 as operands by using a prefix.



.JOP

No Operation (PREFIX)

NOP

Operation

no operation

Assembler Syntax

NOP

Description

NOP

This instruction is encoded as a one-word prefix inside the set, or alone. If the NOP is the only instruction in the execution set, it takes one cycle to execute although no operation is done. This is useful in case delays are needed in a program for various reasons (for example, to account for pipeline delays). The NOP instruction is not dispatched to any execution unit. As a prefix, it is identified by the dispatcher and is not dispatched further.

If grouped with other instructions (as an intra-group NOP), it functions as a program place-holder.

In a few isolated cases, the assembler adds this instruction inside an execution set to help arrange the instructions within that set for proper dispatching. These cases are implementation-dependent. The assembler issues a warning to make the user aware of the event.

Status and conditions that Affect Instruction

None.

Status and Conditions Changed by Instruction

None.

Example

nop

Instruction Formats and Opcodes

Instruction	Words	Cycles	Type	Opcode
NOP	1	1	4	<div style="display: flex; justify-content: space-between; align-items: center;"> 15 8 7 0 </div> <div style="border: 1px solid black; padding: 2px; display: flex; justify-content: space-between; align-items: center;"> 1 0 0 1 0 0 0 0 1 1 0 0 0 0 0 0 </div>

Operation

$\sim Da \rightarrow Dn$

Assembler Syntax

NOT Da, Dn

Description

NOT Da, Dn

Replaces the contents of the destination data register (Dn) with the 40-bit one's complement of the source data register (Da).

Status and Conditions that Affect Instruction

None.

Status and Conditions Changed by Instruction

Register Address	Bit Name	Description
Ln	L	Clears the Ln bit in the destination register.

Example

not d4, d5

Register/Memory Address	Before	After
D4	\$FF FFFF FFFB	
L5:D5		\$0:\$00 0000 0004

Instruction Formats and Opcodes

Instruction	Words	Cycles	Type	Opcode
NOT Da, Dn	1	1	2	<div style="display: flex; justify-content: space-between; align-items: center;"> 15 8 7 0 </div> <div style="border: 1px solid black; padding: 2px; display: flex; justify-content: space-between; align-items: center;"> 1 1 0 1 1 0 F F F 0 0 0 0 J J J </div>

Instruction Fields

Dn	FFF	Single Source/Destination Data Register						
	000	D0	010	D2	100	D4	110	D6
	001	D1	011	D3	101	D5	111	D7

Note: This instruction can specify D8-D15 as operands by using a prefix.

**JJJ****Single Source Data Register**

000	D0	010	D2	100	D4	110	D6
001	D1	011	D3	101	D5	111	D7

Note: This instruction can specify D8-D15 as operands by using a prefix.

Operation

~DR.L → DR.L

~DR.H → DR.H

Assembler Syntax

NOT DR.L

NOT DR.H

Description

NOT DR.L

Inverts the LP of a source data or address register (DR). The other bits are unchanged. This instruction is assembler-mapped to BMCHG DR.L with the full mask enabled.

NOT DR.H

Inverts the HP of a source data or address register (DR). The other bits are unchanged. This instruction is assembler-mapped to BMCHG DR.H with the full mask enabled.

Status and Conditions that Affect Instruction

None.

Status and Conditions Changed by Instruction

Register Address	Bit Name	Description
Ln	L	Clears the Ln bit in the destination registers.

Example

not D0.L

Register/Memory Address	Before	After
L0:D0	\$1:\$00 3FF2 FFFB	\$0:\$00 3FF2 0004

Operation

 $\sim(R) \rightarrow (R)$
 $\sim(SP-u5) \rightarrow (SP-u5)$
 $\sim(SP+s16) \rightarrow (SP+s16)$
 $\sim(a16) \rightarrow (a16)$

Assembler Syntax

`NOT.W (Rn)`
`NOT.W (SP-u5) {0 ≤ u5 < 64, W}`
`NOT.W (SP+s16) {-215 ≤ s16 < 215, W}`
`NOT.W (a16) {0 ≤ a16 < 216, W}`

Description

These operations read from memory, invert the retrieved value, and write the new value back to the same memory address, resulting in two memory accesses.

The absolute addresses, offsets, and address register values must be word-aligned.

NOT.W (Rn)

Replaces the contents of a memory address pointed to by an address register (Rn) with its complement. This instruction is assembler-mapped to `BMCHG.W #FFFFFF,(Rn)`. The full mask is enabled.

NOT.W (SP-u5)

Replaces the contents of a memory address pointed to by the active stack pointer (SP) minus a 5-bit unsigned immediate value with its complement. This instruction is assembler-mapped to `BMCHG.W #FFFFFF,(SP-u5)`. The full mask is enabled.

NOT.W (SP+s16)

Replaces the contents of a memory address pointed to by the active stack pointer (SP) offset by a 16-bit signed immediate value with its complement. This instruction is assembler-mapped to `BMCHG.W #FFFFFF,(SP+s16)`. The full mask is enabled.

NOT.W (a16)

Replaces the contents of a memory address pointed to by a 16-bit unsigned absolute address with its complement. This instruction is assembler-mapped to `BMCHG.W #FFFFFF,(a16)`. The full mask is enabled.

Status and Conditions that Affect Instruction

Register Address	Bit Name	Description
SR[18]	EXP	Determines which stack pointer is used when the stack pointer is an operand. Otherwise, the instruction is not affected by SR.

Status and Conditions Changed by Instruction

None.



xample

not.w (r1)

Register/Memory Address	Before	After
R1	\$0000 0050	
(\$50)	\$FFFFB	\$0004

Instruction Formats and Opcodes

Instruction	Words	Cycles	Type	Opcode																
NOT.W (Rn)	2	2	3	<table border="1"> <tr> <td>15</td> <td>8</td> <td>7</td> <td>0</td> </tr> <tr> <td>0 0 0 1 0 0 1 0</td> <td>1 1 1 0 1 R R R</td> <td></td> <td></td> </tr> <tr> <td>1 0 1 1 1 1 1 1</td> <td>1 1 1 1 1 1 1 1</td> <td></td> <td></td> </tr> </table>	15	8	7	0	0 0 0 1 0 0 1 0	1 1 1 0 1 R R R			1 0 1 1 1 1 1 1	1 1 1 1 1 1 1 1						
15	8	7	0																	
0 0 0 1 0 0 1 0	1 1 1 0 1 R R R																			
1 0 1 1 1 1 1 1	1 1 1 1 1 1 1 1																			
NOT.W (SP-u5)	2	3	3	<table border="1"> <tr> <td>15</td> <td>8</td> <td>7</td> <td>0</td> </tr> <tr> <td>0 0 0 0 0 0 1 0</td> <td>1 1 1 A A A A A</td> <td></td> <td></td> </tr> <tr> <td>1 0 1 1 1 1 1 1</td> <td>1 1 1 1 1 1 1 1</td> <td></td> <td></td> </tr> </table>	15	8	7	0	0 0 0 0 0 0 1 0	1 1 1 A A A A A			1 0 1 1 1 1 1 1	1 1 1 1 1 1 1 1						
15	8	7	0																	
0 0 0 0 0 0 1 0	1 1 1 A A A A A																			
1 0 1 1 1 1 1 1	1 1 1 1 1 1 1 1																			
NOT.W (SP+s16)	3	3	3	<table border="1"> <tr> <td>15</td> <td>8</td> <td>7</td> <td>0</td> </tr> <tr> <td>0 0 1 1 1 0 1 0</td> <td>A A A 1 1 0 1 1</td> <td></td> <td></td> </tr> <tr> <td>0 0 1 A A A A A</td> <td>A A A A A A A A</td> <td></td> <td></td> </tr> <tr> <td>1 0 1 1 1 1 1 1</td> <td>1 1 1 1 1 1 1 1</td> <td></td> <td></td> </tr> </table>	15	8	7	0	0 0 1 1 1 0 1 0	A A A 1 1 0 1 1			0 0 1 A A A A A	A A A A A A A A			1 0 1 1 1 1 1 1	1 1 1 1 1 1 1 1		
15	8	7	0																	
0 0 1 1 1 0 1 0	A A A 1 1 0 1 1																			
0 0 1 A A A A A	A A A A A A A A																			
1 0 1 1 1 1 1 1	1 1 1 1 1 1 1 1																			
NOT.W (a16)	3	2	3	<table border="1"> <tr> <td>15</td> <td>8</td> <td>7</td> <td>0</td> </tr> <tr> <td>0 0 1 1 1 0 1 0</td> <td>A A A 1 1 0 0 1</td> <td></td> <td></td> </tr> <tr> <td>0 0 1 A A A A A</td> <td>A A A A A A A A</td> <td></td> <td></td> </tr> <tr> <td>1 0 1 1 1 1 1 1</td> <td>1 1 1 1 1 1 1 1</td> <td></td> <td></td> </tr> </table>	15	8	7	0	0 0 1 1 1 0 1 0	A A A 1 1 0 0 1			0 0 1 A A A A A	A A A A A A A A			1 0 1 1 1 1 1 1	1 1 1 1 1 1 1 1		
15	8	7	0																	
0 0 1 1 1 0 1 0	A A A 1 1 0 0 1																			
0 0 1 A A A A A	A A A A A A A A																			
1 0 1 1 1 1 1 1	1 1 1 1 1 1 1 1																			

Instruction Fields

Rn	RRR	Address Register						
	000	R0	010	R2	100	R4	110	R6
	001	R1	011	R3	101	R5	111	R7

Note: This instruction can specify R8-R15 as operands by using a high register prefix.

- a16** AAAAAAAAAAAAAAAAAA 16-bit unsigned absolute address
- u5** AAAAA0 Unsigned 5-bit address offset
- s16** AAAAAAAAAAAAAAAAAA Signed 16-bit SP address offset

Operation

Da | Dn → Dn

Assembler Syntax

OR Da, Dn

Description

OR Da, Dn

Performs a bitwise inclusive OR of two data registers (Da and Dn) and stores the result in the second data register (Dn). This is a full 40-bit operation.

Status and Conditions that Affect Instruction

None.

Status and Conditions Changed by Instruction

Register Address	Bit Name	Description
Ln	L	Clears the Ln bit in the destination registers.

Example

or d3, d0

Register/Memory Address	Before	After
D3	\$E0 0007 0005	
L0:D0	\$0:\$50 0003 0008	\$0:\$F0 0007 000F

```

1110 ---- 0111 ---- 0101
or 0101 ---- 0011 ---- 1000
-----
1111 ---- 0111 ---- 1111
    
```

Instruction Formats and Opcodes

Instruction	Words	Cycles	Type	Opcode
OR Da, Dn	1	1	2	<div style="display: flex; justify-content: space-between; align-items: center;"> 15 8 7 0 </div> <div style="border: 1px solid black; padding: 2px; display: flex; justify-content: space-between; align-items: center;"> 1 1 0 1 1 1 F F F 0 0 1 1 J J J </div>

Instruction Fields

Da	JJJ	Single Source Data Register					
000	D0	010	D2	100	D4	110	D6
001	D1	011	D3	101	D5	111	D7

Note: This instruction can specify D8-D15 as operands by using a prefix.

FFF **Single Source/Destination Data Register**

000	D0	010	D2	100	D4	110	D6
001	D1	011	D3	101	D5	111	D7

Note: This instruction can specify D8-D15 as operands by using a prefix.

Operation
 $\#u16 \mid DR.L \rightarrow DR.L$
 $\#u16 \mid DR.H \rightarrow DR.H$
Assembler Syntax
 $OR \ \#u16, DR.L \ \{0 \leq u16 < 2^{16}\}$
 $OR \ \#u16, DR.H \ \{0 \leq u16 < 2^{16}\}$
Description
OR #u16,DR.L

Performs a bitwise inclusive OR of an immediate value with the LP of a data or address register (DR). It then stores the result in the LP of the destination data or address register (DR). The other register bits are not affected. This instruction is assembler-mapped to BMSET #u16,DR.L with the immediate value.

OR #u16,DR.H

Performs a bitwise inclusive OR of an immediate value with the HP of a data or address register (DR). It then stores the result in the HP of the destination data or address register (DR). The other register bits are not affected. This instruction is assembler-mapped to BMSET #u16,DR.H with the immediate value.

Status and Conditions that Affect Instruction

None.

Status and Conditions Changed by Instruction

Register Address	Bit Name	Description
Ln	L	Clears the Ln bit in the destination registers.

Example

```
or #0f0a,d0.l
```

Register/Memory Address	Before	After
Immediate	\$0F0A	
D0	\$1:\$00 ACBD F065	\$0:\$00 ACBD FF6F

```

0000 1111 0000 1010
or 1111 0000 0110 0101
-----
1111 1111 0110 1111

```



Instruction Formats and Opcodes

Instruction	Words	Cycles	Type	Opcode												
OR #u16,DR.L	2	2	3	<table border="1"> <tr> <td style="text-align: right;">15</td> <td style="text-align: center;">8</td> <td style="text-align: center;">7</td> <td style="text-align: right;">0</td> </tr> <tr> <td>0 0 0 0 1 0 0 1</td> <td>i</td> <td>i</td> <td>i 0 H H H H</td> </tr> <tr> <td>1 0 1 i i i i i</td> <td>i</td> <td>i</td> <td>i i i i i</td> </tr> </table>	15	8	7	0	0 0 0 0 1 0 0 1	i	i	i 0 H H H H	1 0 1 i i i i i	i	i	i i i i i
15	8	7	0													
0 0 0 0 1 0 0 1	i	i	i 0 H H H H													
1 0 1 i i i i i	i	i	i i i i i													
OR #u16,DR.H	2	2	3	<table border="1"> <tr> <td style="text-align: right;">15</td> <td style="text-align: center;">8</td> <td style="text-align: center;">7</td> <td style="text-align: right;">0</td> </tr> <tr> <td>0 0 0 0 1 0 0 1</td> <td>i</td> <td>i</td> <td>i 1 H H H H</td> </tr> <tr> <td>1 0 1 i i i i i</td> <td>i</td> <td>i</td> <td>i i i i i</td> </tr> </table>	15	8	7	0	0 0 0 0 1 0 0 1	i	i	i 1 H H H H	1 0 1 i i i i i	i	i	i i i i i
15	8	7	0													
0 0 0 0 1 0 0 1	i	i	i 1 H H H H													
1 0 1 i i i i i	i	i	i i i i i													

Instruction Fields

DR	HHHH	Data/Address Register					
0000	D0	0100	D4	1000	R0	1100	R4
0001	D1	0101	D5	1001	R1	1101	R5
0010	D2	0110	D6	1010	R2	1110	R6
0011	D3	0111	D7	1011	R3	1111	R7

Note: This instruction can specify D8-D15 or R8-R15 as operands by using a high register prefix.

#u16 iiii...iiiiii 16-bit unsigned immediate data

OR.W Bitwise OR on a 16-Bit Operand in Memory (BMU) OR.W

Operation

Assembler Syntax

#u16 (R) → (R)	OR.W #u16, (Rn) {0 ≤ u16 < 2 ¹⁶ }
#u16 (SP-u5) → (SP-u5)	OR.W #u16, (SP-u5) {0 ≤ u16 < 2 ¹⁶ } {0 ≤ u5 < 64, W}
#u16 (SP+s16) → (SP+s16)	OR.W #u16, (SP+s16) {0 ≤ u16 < 2 ¹⁶ } {-2 ¹⁵ ≤ s16 < 2 ¹⁵ , W}
#u16 (a16) → (a16)	OR.W #u16, (a16) {0 ≤ u16 < 2 ¹⁶ } {0 ≤ a16 < 2 ¹⁶ , W}

Description

These operations read from memory, modify the retrieved value, and write the new value back to the same memory address, resulting in two memory accesses.

The absolute addresses, offsets, and address register values must be word-aligned.

OR.W #u16,(Rn)

Performs a bitwise inclusive OR of an immediate unsigned word with the contents of a memory address pointed to by an address register (Rn). It then stores the result in that memory address. This instruction is assembler-mapped to BMSET.W #u16,(Rn) with the immediate value.

OR.W #u16,(SP-u5)

Performs a bitwise inclusive OR of an immediate unsigned word with the contents of a memory address pointed to by the active stack pointer (SP) minus an unsigned 5-bit offset. It then stores the result in the memory address. This instruction is assembler-mapped to BMSET.W #u16,(SP-u5) with the immediate value.

OR.W #u16,(SP+s16)

Performs a bitwise inclusive OR of an immediate unsigned word with the contents of a memory address pointed to by the active stack pointer (SP) plus by a signed 16-bit offset. It then stores the result in the memory address. This instruction is assembler-mapped to BMSET.W #u16,(SP+s16) with the immediate value.

OR.W #u16,(a16)

Performs a bitwise inclusive OR of an immediate unsigned word with the contents of a 16-bit absolute memory address. It then stores the result in the memory location. This instruction is assembler-mapped to BMSET.W #u16,(a16) with the immediate value.

Status and Conditions that Affect Instruction

Register Address	Bit Name	Description
SR[18]	EXP	Determines which stack pointer is used when the stack pointer is an operand. Otherwise, the instruction is not affected by SR.

Status and Conditions Changed by Instruction

None.

Example

or.w #f01a, (r1)

Register/Memory Address	Before	After
Immediate	\$F01A	
R1	\$0000 0050	
(\$0050)	\$1235	\$F23F

1111 0000 0001 1010
 or 0001 0010 0011 0101
 1111 0010 0011 1111

Instruction Formats and Opcodes

Instruction	Words	Cycles	Type	Opcode																
OR.W #u16, (Rn)	2	2	3	<table border="1"> <tr> <td>15</td> <td>8</td> <td>7</td> <td>0</td> </tr> <tr> <td>0 0 0 1 0 0 0 1</td> <td>i i i 0 1 R R R</td> <td></td> <td></td> </tr> <tr> <td>1 0 1 i i i i i</td> <td>i i i i i i i i</td> <td></td> <td></td> </tr> </table>	15	8	7	0	0 0 0 1 0 0 0 1	i i i 0 1 R R R			1 0 1 i i i i i	i i i i i i i i						
15	8	7	0																	
0 0 0 1 0 0 0 1	i i i 0 1 R R R																			
1 0 1 i i i i i	i i i i i i i i																			
OR.W #u16, (SP-u5)	2	3	3	<table border="1"> <tr> <td>15</td> <td>8</td> <td>7</td> <td>0</td> </tr> <tr> <td>0 0 0 0 0 0 0 1</td> <td>i i i A A A A A</td> <td></td> <td></td> </tr> <tr> <td>1 0 1 i i i i i</td> <td>i i i i i i i i</td> <td></td> <td></td> </tr> </table>	15	8	7	0	0 0 0 0 0 0 0 1	i i i A A A A A			1 0 1 i i i i i	i i i i i i i i						
15	8	7	0																	
0 0 0 0 0 0 0 1	i i i A A A A A																			
1 0 1 i i i i i	i i i i i i i i																			
OR.W #u16, (SP+s16)	3	3	3	<table border="1"> <tr> <td>15</td> <td>8</td> <td>7</td> <td>0</td> </tr> <tr> <td>0 0 1 1 1 0 0 1</td> <td>A A A i i 0 1 1</td> <td></td> <td></td> </tr> <tr> <td>0 0 1 A A A A A</td> <td>A A A A A A A A</td> <td></td> <td></td> </tr> <tr> <td>1 0 i i i i i i</td> <td>i i i i i i i i</td> <td></td> <td></td> </tr> </table>	15	8	7	0	0 0 1 1 1 0 0 1	A A A i i 0 1 1			0 0 1 A A A A A	A A A A A A A A			1 0 i i i i i i	i i i i i i i i		
15	8	7	0																	
0 0 1 1 1 0 0 1	A A A i i 0 1 1																			
0 0 1 A A A A A	A A A A A A A A																			
1 0 i i i i i i	i i i i i i i i																			
OR.W #u16, (a16)	3	2	3	<table border="1"> <tr> <td>15</td> <td>8</td> <td>7</td> <td>0</td> </tr> <tr> <td>0 0 1 1 1 0 0 1</td> <td>A A A i i 0 0 1</td> <td></td> <td></td> </tr> <tr> <td>0 0 1 A A A A A</td> <td>A A A A A A A A</td> <td></td> <td></td> </tr> <tr> <td>1 0 i i i i i i</td> <td>i i i i i i i i</td> <td></td> <td></td> </tr> </table>	15	8	7	0	0 0 1 1 1 0 0 1	A A A i i 0 0 1			0 0 1 A A A A A	A A A A A A A A			1 0 i i i i i i	i i i i i i i i		
15	8	7	0																	
0 0 1 1 1 0 0 1	A A A i i 0 0 1																			
0 0 1 A A A A A	A A A A A A A A																			
1 0 i i i i i i	i i i i i i i i																			

Instruction Fields

Rn	RRR	Address Register					
000	R0	010	R2	100	R4	110	R6
001	R1	011	R3	101	R5	111	R7

Note: This instruction can specify R8-R15 as operands by using a high register prefix.

#u16	iiiiiiiiiiiiii	16-bit unsigned immediate data
a16	AAAAAAAAAAAAAAAA	16-bit unsigned absolute address
u5	AAAAA0	Unsigned 5-bit SP address offset



- 6

AAAAAAAAAAAAAAAA

Signed 16-bit SP address offset

POP

Pop a Register from the Software Stack (AGU)

POP

Operation

 $(SP - 8) \rightarrow De; SP - 8 \rightarrow SP$
 $(SP - 4) \rightarrow Do; SP - 8 \rightarrow SP$

Assembler Syntax

POP De

POP Do

Description

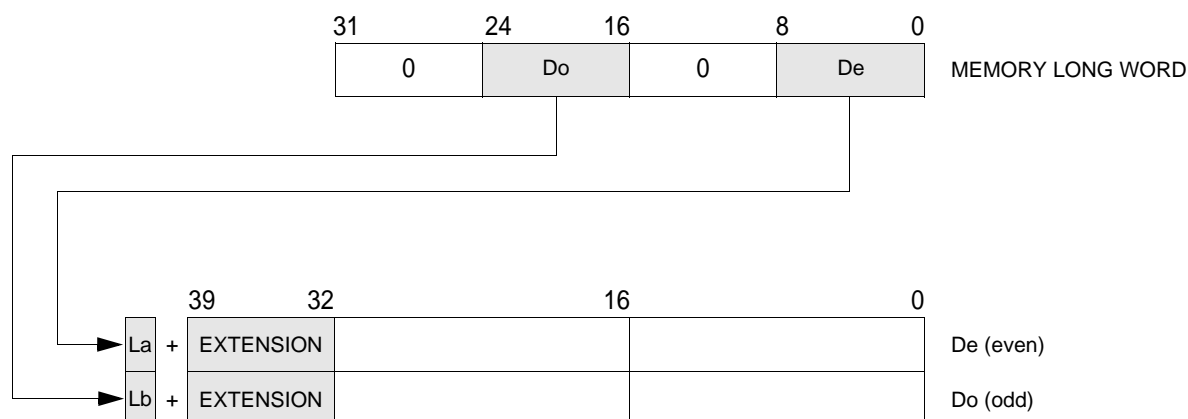
These operations read the memory address pointed to by the active stack pointer (SP) into an even or odd register (De or Do) and adjust SP. All memory accesses are 32-bit long words. The registers are divided into two groups (even and odd) which determines the memory offset relative to the SP of the data being read. It is important to pop registers in the same register grouping by which they were pushed. For example, after the sequence "push d1" and "pop d3," d3 will hold the data originally in d1. However, "push d0" and "pop d1" will not do the same because d0 and d1 are not in the same register group.

One or two POP instructions can appear in an execution set. In both cases, SP is decremented only once by 8. When two POP instructions are grouped together in an execution set, each must be in a different register group.

If the register is a DALU register, bits [39:32] of the destination are sign-extended from bit 31 and the Ln bit is cleared. Hence, in order to restore a full data register, the extension should be popped last.

Extensions of data registers (with the associated Ln bits) are special. Extensions of even and odd registers are read from bits [8:0] and [24:16] of the long data word, respectively, both for single register and register pair operations (see the figure below).

Note: For proper data register restoration, extensions that were pushed as a pair should be popped as a pair. Extensions pushed as single registers should be popped as single registers.

**POP De**

Restores data register extension pairs, even registers, and loop start registers from the stack. Data register extension pairs are popped the same as even numbered registers.

POP Do

Restores modifier control, odd registers, and loop counter registers from the stack.

Status and Conditions that Affect Instruction

Register Address	Bit Name	Description
SR[18]	EXP	Determines which stack pointer used, and which execution working mode .

Status and Conditions Changed By Instruction

Register Address	Bit Name	Description
Ln	L	Pops of extensions restore the Ln bit in the destination register. Pops to data registers clear the Ln bit.

Example

```
pop d3
```

Register/Memory Address	Before	After
SR	\$00E00000	
NSP	\$00000010	\$00000008
\$0000000C	\$2E03FF4E	
L3:D3		\$0:\$002E03FF4E

Instruction Formats and Opcodes

Instruction	Words	Cycles ¹	Type	Opcode
POP De	1	1	4	<div style="display: flex; justify-content: space-between; margin: 0;"> 15 8 7 0 </div> <div style="border: 1px solid black; padding: 2px; display: flex; justify-content: space-between;"> 1 0 0 1 E E E 0 0 0 1 E 0 0 E 1 </div>
POP Do	1	1	4	<div style="display: flex; justify-content: space-between; margin: 0;"> 15 8 7 0 </div> <div style="border: 1px solid black; padding: 2px; display: flex; justify-content: space-between;"> 1 0 0 1 e e e 1 0 0 1 e 0 0 e 1 </div>

Note 1: An extra cycle is added if the shadow SP is not valid when the POP instruction is executed. See [Section 5.5.4, “Shadow Stack Pointer Registers,”](#)

Instruction Fields

EEEE Extension Pairs, Even Registers, and Loop Start Registers

00000	D0	01000	D4	10000	R0	11000	R4
00001	B0	01001	B4	10001	N0	11001	M0
00010	—	01010	—	10010	SA0	11010	SA2
00011	D0.E	01011	D4.E	10011	D0.E:D1.E	11011	D4.E:D5.E
00100	D2	01100	D6	10100	R2	11100	R6
00101	B2	01101	B6	10101	N2	11101	M2
00110	-	01110	—	10110	SA1	11110	SA3
00111	D2.E	01111	D6.E	10111	D2.E:D3.E	11111	D6.E:D7.E

Note: If registers D8–D15 or R8–R15 are accessed instead of D0–D7 or R0–R7, a prefix is used.

Do

eeee Modifier Control, Odd Registers, and Loop Counter Registers

00000	D1	01000	D5	10000	R1	11000	R5
00001	B1	01001	B5	10001	N1	11001	M1
00010	VBA	01010	SR	10010	LC0	11010	LC2
00011	D1.E	01011	D5.E	10011	—	11011	—
00100	D3	01100	D7	10100	R3	11100	R7
00101	B3	01101	B7	10101	N3	11101	M3
00110	-	01110	MCTL	10110	LC1	11110	LC3
00111	D3.E	01111	D7.E	10111	—	11111	—

Note: If registers D8–D15 or R8–R15 are accessed instead of D0–D7 or R0–R7, a prefix is used.

Operation

$(NSP - 8) \rightarrow De$; $NSP - 8 \rightarrow NSP$

$(NSP - 4) \rightarrow Do$; $NSP - 8 \rightarrow NSP$

Assembler Syntax

POPN De

POPN Do

Description

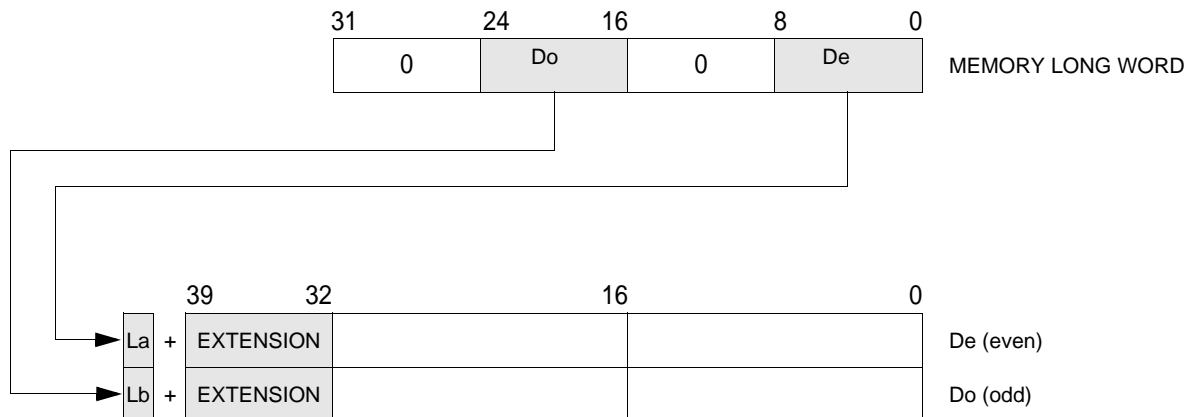
These operations read the memory address pointed to by the normal stack pointer (NSP) into an even or odd register (De or Do) and adjust NSP regardless of the state of the exception (EXP) bit. All memory accesses are 32-bit long words. The registers are divided into two groups (even and odd) which determines the memory offset relative to NSP of the data being read. It is important to pop registers in the same register grouping by which they were pushed. For example, after the sequence "pushn d1" and "popn d3," d3 will hold the data originally in d1. However, "pushn d0" and "popn d1" will not do the same because d0 and d1 are not in the same register group.

One or two POP instructions can appear in an execution set. In both cases, NSP is decremented only once by 8. When two POP instructions are grouped together in an execution set, each must be in a different register group.

If the register is a DALU register, bits [39:32] of the destination are sign-extended from bit 31 and the Ln bit is cleared. Hence, in order to restore a full data register, the extension should be popped last.

Extensions of data registers (with the associated Ln bits) are special. Extensions of even and odd registers are read from bits [8:0] and [24:16] of the long data word, respectively, both for single register and register pair operations (see the figure below).

Note: For proper data register restoration, extensions that were pushed as a pair should be popped as a pair. Extensions pushed as single registers should be popped as single registers.



rOPN De

Restores data register extension pairs, even registers, and loop start registers from the normal stack. Data register extension pairs are popped the same as even numbered registers.

POPn Do

Restores modifier control, odd registers, and loop counter registers from the normal stack.

Status and Conditions that Affect Instruction

Register Address	Bit Name	Description
SR[18]	EXP	Determines execution working mode.

Status and Conditions Changed By Instruction

Register Address	Bit Name	Description
Ln	L	Pops of extensions restore the Ln bit in the destination register. Pops to data registers clear the Ln bit.

Example

```
popn d6.e:d7.e
```

Register/Memory Address	Before	After
NSP	\$00000010	\$00000008
\$00000008	\$000000FF	
L6:D6		\$0:\$FFF0000000
L7:D7		\$0:\$0000000000

Instruction Formats and Opcodes

Instruction	Words	Cycles ¹	Type	Opcode																				
POPn De	1	1	4	<table border="1"> <tr> <td>15</td> <td>8</td> <td>7</td> <td>0</td> </tr> <tr> <td>1</td><td>0</td><td>0</td><td>1</td> <td>E</td><td>E</td><td>E</td><td>0</td> <td>0</td><td>0</td><td>1</td><td>E</td><td>0</td><td>1</td><td>E</td><td>1</td> </tr> </table>	15	8	7	0	1	0	0	1	E	E	E	0	0	0	1	E	0	1	E	1
15	8	7	0																					
1	0	0	1	E	E	E	0	0	0	1	E	0	1	E	1									
POPn Do	1	1	4	<table border="1"> <tr> <td>15</td> <td>8</td> <td>7</td> <td>0</td> </tr> <tr> <td>1</td><td>0</td><td>0</td><td>1</td> <td>e</td><td>e</td><td>e</td><td>1</td> <td>0</td><td>0</td><td>1</td><td>e</td><td>0</td><td>1</td><td>e</td><td>1</td> </tr> </table>	15	8	7	0	1	0	0	1	e	e	e	1	0	0	1	e	0	1	e	1
15	8	7	0																					
1	0	0	1	e	e	e	1	0	0	1	e	0	1	e	1									

Note 1: An extra cycle is added if the shadow SP is not valid when the POP instruction is executed. See Section 5.3.3, "Shadow Stack Pointer Registers."

Instruction Fields

EEEE Extension Pairs, Even Registers, and Loop Start Registers

00000	D0	01000	D4	10000	R0	11000	R4
00001	B0	01001	B4	10001	N0	11001	M0
00010	—	01010	—	10010	SA0	11010	SA2
00011	D0.E	01011	D4.E	10011	D0.E:D1.E	11011	D4.E:D5.E
00100	D2	01100	D6	10100	R2	11100	R6
00101	B2	01101	B6	10101	N2	11101	M2
00110	-	01110	—	10110	SA1	11110	SA3
00111	D2.E	01111	D6.E	10111	D2.E:D3.E	11111	D6.E:D7.E

Note: If registers D8–D15 or R8–R15 are accessed instead of D0–D7 or R0–R7, a prefix is used.

Do
eeee Modifier Control, Odd Registers, and Loop Counter Registers

00000	D1	01000	D5	10000	R1	11000	R5
00001	B1	01001	B5	10001	N1	11001	M1
00010	VBA	01010	SR	10010	LC0	11010	LC2
00011	D1.E	01011	D5.E	10011	—	11011	—
00100	D3	01100	D7	10100	R3	11100	R7
00101	B3	01101	B7	10101	N3	11101	M3
00110	-	01110	MCTL	10110	LC1	11110	LC3
00111	D3.E	01111	D7.E	10111	—	11111	—

Note: If registers D8–D15 or R8–R15 are accessed instead of D0–D7 or R0–R7, a prefix is used.

PUSH Push a Register onto the Software Stack (AGU) **PUSH**

Operation

$De \rightarrow (SP); SP + 8 \rightarrow SP$

$Do \rightarrow (SP + 4); SP + 8 \rightarrow SP$

Assembler Syntax

`PUSH De`

`PUSH Do`

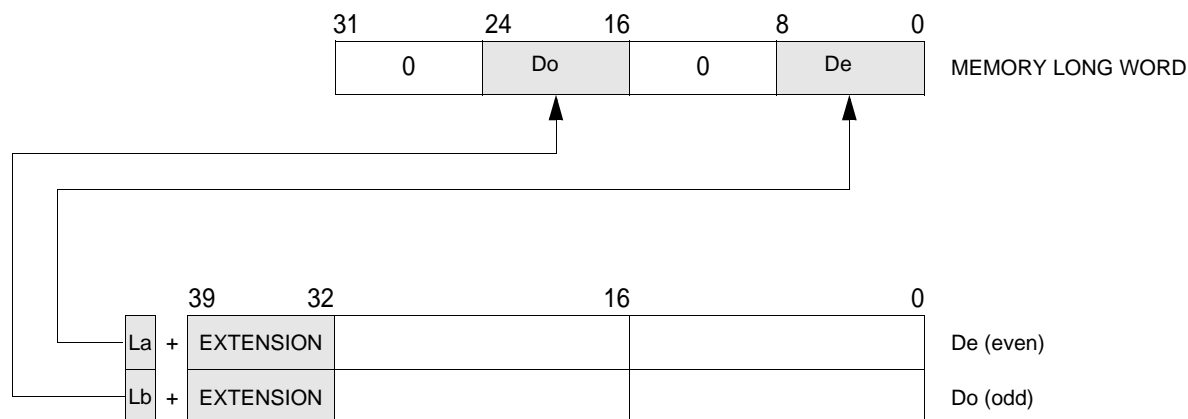
Description

These operations move an even or odd register (*De* or *Do*) to the active stack in memory and adjust *SP*. All memory accesses are 32-bit long words. The registers are divided into two groups (even and odd) which determines the memory offset relative to *SP* of the data being written. It is important to pop registers in the same register grouping by which they were pushed. For example, after the sequence "push d1" and "pop d3," d3 will hold the data originally in d1. However, "push d0" and "pop d1" will not do the same because d0 and d1 are not in the same register group.

One or two **PUSH** instructions can appear in an execution set. In both cases, *SP* is incremented only once by 8. When two **PUSH** instructions are grouped together in an execution set, each must be in a different register group.

Extensions of data registers (with the associated *Ln* bits) are special. Extensions of even and odd registers are written to bits [8:0] and [24:16] of the long data word, respectively, both for single register and register pair operations (see the figure below).

Note: For proper data register restoration, extensions that were pushed as a pair should be popped as a pair. Extensions pushed as single registers should be popped as single registers.



PUSH De

Pushes data register extension pairs, even registers, and loop start registers onto the current stack. Data register extension pairs are pushed the same as even numbered registers.

PUSH Do

Pushes modifier control, odd registers, and loop counter registers onto the current stack.

Status and Conditions that Affect Instruction

Register Address	Bit Name	Description
SR[18]	EXP	Determines which stack pointer used, and execution working mode.

Status and Conditions Changed By Instruction

None

Example

```
push d0.e:d1.e
```

Register/Memory Address	Before	After
SR	\$00E40000	
ESP	\$00000000	\$00000008
(\$00000000)	\$00000000	\$000000FF
L0:D0	\$0:\$FF89ABCDEF	
L1:D1	\$0:\$0001234567	

Instruction Formats and Opcodes

Instruction	Words	Cycles	Type	Opcode								
PUSH De	1	1	4	<table border="1"> <tr> <td>15</td> <td>8</td> <td>7</td> <td>0</td> </tr> <tr> <td>1 0 0 1 E E E 0</td> <td>0 0 1 E 0 0 E 0</td> <td></td> <td></td> </tr> </table>	15	8	7	0	1 0 0 1 E E E 0	0 0 1 E 0 0 E 0		
15	8	7	0									
1 0 0 1 E E E 0	0 0 1 E 0 0 E 0											
PUSH Do	1	1	4	<table border="1"> <tr> <td>15</td> <td>8</td> <td>7</td> <td>0</td> </tr> <tr> <td>1 0 0 1 e e e 1</td> <td>0 0 1 e 0 0 e 0</td> <td></td> <td></td> </tr> </table>	15	8	7	0	1 0 0 1 e e e 1	0 0 1 e 0 0 e 0		
15	8	7	0									
1 0 0 1 e e e 1	0 0 1 e 0 0 e 0											



struction Fields

De EEEEE Extension Pairs, Even Registers, and Loop Start Registers

00000	D0	01000	D4	10000	R0	11000	R4
00001	B0	01001	B4	10001	N0	11001	M0
00010	—	01010	—	10010	SA0	11010	SA2
00011	D0.E	01011	D4.E	10011	D0.E:D1.E	11011	D4.E:D5.E
00100	D2	01100	D6	10100	R2	11100	R6
00101	B2	01101	B6	10101	N2	11101	M2
00110	-	01110	—	10110	SA1	11110	SA3
00111	D2.E	01111	D6.E	10111	D2.E:D3.E	11111	D6.E:D7.E

Note: If registers D8–D15 or R8–R15 are accessed instead of D0–D7 or R0–R7, a prefix is used.

Do eeeee Modifier Control, Odd Registers, and Loop Counter Registers

00000	D1	01000	D5	10000	R1	11000	R5
00001	B1	01001	B5	10001	N1	11001	M1
00010	VBA	01010	SR	10010	LC0	11010	LC2
00011	D1.E	01011	D5.E	10011	—	11011	—
00100	D3	01100	D7	10100	R3	11100	R7
00101	B3	01101	B7	10101	N3	11101	M3
00110	-	01110	MCTL	10110	LC1	11110	LC3
00111	D3.E	01111	D7.E	10111	—	11111	—

Note: If registers D8–D15 or R8–R15 are accessed instead of D0–D7 or R0–R7, a prefix is used.

PUSHN Push a Register onto the Software Stack Using the Normal Stack Pointer (AGU) PUSHN

Operation

De → (NSP); NSP + 8 → NSP

Do → (NSP + 4); NSP + 8 → NSP

Assembler Syntax

PUSHN De

PUSHN Do

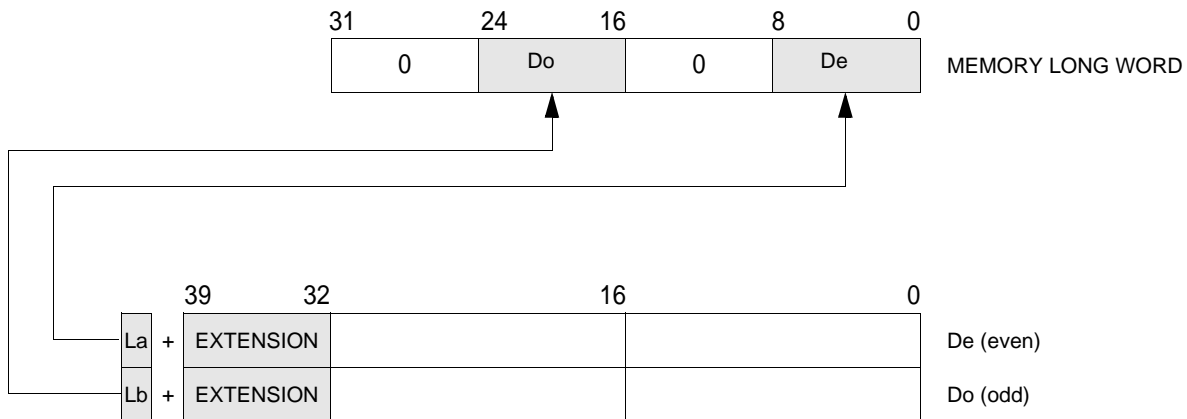
Description

These operations move an even or odd register (De or Do) to the normal stack in memory and adjusts the NSP, regardless of the state of the exception (EXP) bit. All memory accesses are 32-bit long words. The registers are divided into two groups (even and odd) which determines the memory offset relative to NSP of the data being written. It is important to pop registers in the same register grouping by which they were pushed. For example, after the sequence "pushn d1" and "popn d3," d3 will hold the data originally in d1. However, "pushn d0" and "popn d1" will not do the same because d0 and d1 are not in the same register group.

One or two PUSHN instructions can appear in an execution set. In both cases, NSP is incremented only once by 8. When two PUSHN instructions are grouped together in an execution set, each must be in a different register group.

Extensions of data registers (with the associated Ln bits) are special. Extensions of even and odd registers are written to bits [8:0] and [24:16] of the long data word, respectively, both for single register and register pair operations (see the figure below).

Note: For proper data register restoration, extensions that were pushed as a pair should be popped as a pair. Extensions pushed as single registers should be popped as single registers



PUSHN De

Pushes data register extension pairs, even registers, and loop start registers onto the current stack. Data register extension pairs are pushed the same as even numbered registers.

PUSHN Do

Pushes modifier control, odd registers, and loop counter registers onto the current stack.

Status and Conditions that Affect Instruction

Register Address	Bit Name	Description
SR[18]	EXP	Determines execution working mode.

Status and Conditions Changed By Instruction

Register Address	Bit Name	Description
Ln	L	Pops of extensions restore the Ln bit in the destination register. Pops to data registers clear the Ln bit.

Example

`pushn d0.e:d1.e`

Register/Memory Address	Before	After
NSP	\$00000008	\$00000010
L0:D0	\$0:\$FF89ABCDEF	
L1:D1	\$0:\$0001234567	
(\$000008)		\$000000FF

Instruction Formats and Opcodes

Instruction	Words	Cycles	Type	Opcode												
PUSHN De	1	1	4	<table border="1" style="width: 100%; text-align: center;"> <tr> <td style="width: 15%;">15</td> <td style="width: 45%;"></td> <td style="width: 15%;">8</td> <td style="width: 15%;">7</td> <td style="width: 10%;"></td> <td style="width: 10%;">0</td> </tr> <tr> <td>1</td> <td>0 0 1 E E E 0</td> <td>0</td> <td>0</td> <td>1 E 0 1 E 0</td> <td>0</td> </tr> </table>	15		8	7		0	1	0 0 1 E E E 0	0	0	1 E 0 1 E 0	0
15		8	7		0											
1	0 0 1 E E E 0	0	0	1 E 0 1 E 0	0											
PUSHN Do	1	1	4	<table border="1" style="width: 100%; text-align: center;"> <tr> <td style="width: 15%;">15</td> <td style="width: 45%;"></td> <td style="width: 15%;">8</td> <td style="width: 15%;">7</td> <td style="width: 10%;"></td> <td style="width: 10%;">0</td> </tr> <tr> <td>1</td> <td>0 0 1 e e e 1</td> <td>0</td> <td>0</td> <td>1 e 0 1 e 0</td> <td>0</td> </tr> </table>	15		8	7		0	1	0 0 1 e e e 1	0	0	1 e 0 1 e 0	0
15		8	7		0											
1	0 0 1 e e e 1	0	0	1 e 0 1 e 0	0											

Instruction Fields

De EEEEE Extension Pairs, Even Registers, and Loop Start Registers

00000	D0	01000	D4	10000	R0	11000	R4
00001	B0	01001	B4	10001	N0	11001	M0
00010	—	01010	—	10010	SA0	11010	SA2
00011	D0.E	01011	D4.E	10011	D0.E:D1.E	11011	D4.E:D5.E
00100	D2	01100	D6	10100	R2	11100	R6
00101	B2	01101	B6	10101	N2	11101	M2
00110	-	01110	—	10110	SA1	11110	SA3
00111	D2.E	01111	D6.E	10111	D2.E:D3.E	11111	D6.E:D7.E

Note: If registers D8–D15 or R8–R15 are accessed instead of D0–D7 or R0–R7, a prefix is used.

Do eeeee Modifier Control, Odd Registers, and Loop Counter Registers

00000	D1	01000	D5	10000	R1	11000	R5
00001	B1	01001	B5	10001	N1	11001	M1
00010	VBA	01010	SR	10010	LC0	11010	LC2
00011	D1.E	01011	D5.E	10011	—	11011	—
00100	D3	01100	D7	10100	R3	11100	R7
00101	B3	01101	B7	10101	N3	11101	M3
00110	-	01110	MCTL	10110	LC1	11110	LC3
00111	D3.E	01111	D7.E	10111	—	11111	—

Note: If registers D8–D15 or R8–R15 are accessed instead of D0–D7 or R0–R7, a prefix is used.

Operation

Rnd(Da) → Dn

Assembler Syntax

RND Da, Dn

Description

RND Da, Dn

Rounds the 40-bit value in the source data register (Da) and stores the result in the destination data register (Dn). In the round function, the contribution of the least significant bits is rounded into the HP of the destination data register by adding a rounding constant RC to the LS bits of the source data register. The boundary between the LP and HP is determined by the scaling mode bits S0 and S1 in SR. The LSBs of the result are cleared. The number of LSBs cleared is determined by the scaling mode bits in SR. All bits to the right of (including the rounding position) are cleared in the result.

Two types of rounding can be used: convergent rounding (round to the nearest even number) or two's complement rounding. The type of rounding is selected by the rounding mode bit (RM) in SR. The default mode is convergent rounding (SR[3] = 0).

In both rounding modes, a rounding constant (RC) is first added to the source data. The value of the rounding constant added is determined by the scaling mode bits S0 and S1 in SR. A 1 is positioned in the rounding constant aligned with the MSB of the scaled LP. The rounding constant weight is actually equal to half the weight of the scaled HP's LSB.

For two's complement rounding, the scaled LP bits are then truncated. Numbers with an original value of 1/2 in the scaled LP are rounded up, resulting in a small positive bias.

If convergent rounding is used, the result of the addition is tested. If all the bits of the result to the right of (including the rounding position) are cleared, then the bit to the left of the rounding position is cleared in the result, ensuring that the result is even. An even result eliminates the two's complement bias where 1/2 is always rounded up.

See [Section 2.2.2.6, “Rounding Modes,”](#) on page 2-21 for more detailed information.

The following table shows the rounding position (LP MSB) and rounding constant (RC) as determined by the scaling mode bits:

S1	S0	Scaling Mode	Rounding Position	Rounding Constant (RC) Bits				
				39–17	16	15	14	13–0
0	0	No Scaling	15	0...0	0	1	0	0...0
0	1	Scale Down	16	0...0	1	0	0	0...0
1	0	Scale Up	14	0...0	0	0	1	0...0

Status and Conditions that Affect Instruction

Register Address	Bit Name	Description
SR[2]	SM	If set, selects 32-bit arithmetic saturation mode.
SR[3]	RM	Rounding mode
SR[5:4]	S[1:0]	Scaling bits determine which bits in the result are used in the Ln bit calculation and which bits are used in rounding.

Status and Conditions Changed by Instruction

Register Address	Bit Name	Description
Ln	L	If not in arithmetic saturation mode (SR [SM] = 0), calculates and updates the Ln bit in the destination register. If in arithmetic saturation mode (SR [SM] = 1), clears the Ln bit in the destination register.
EMR[2]	DOVF	Set if the result is not representable in 40 bits, or if the result saturates to 32 bits in saturation mode.

Example 1

`rnd d1,d5`

Register/Memory Address	Before	After
SR	\$00E0 0000	
D1	\$00 0000 FFFF	
L5:D5		\$0:\$00 0001 0000
EMR		\$0000 0000

Example 2

`rnd d2,d1`

Register/Memory Address	Before	After
SR	\$00E0 0028	
D2	\$00 CAFE 4000	
L1:D1		\$0:\$00 CAFE 8000
EMR		\$0000 0000

\$CAFE 40001100 1010 1111 1110 **0**100 0000 0000 0000
 After rounding \$CAFE 80001100 1010 1111 1110 **1**000 0000 0000 0000

Scaling up is selected in SR[4-5], and 2's complement rounding is selected in SR[3]. Bit 15 is rounded up because bit 14 = 1.



Instruction Formats and Opcodes

Instruction	Words	Cycles	Type	Opcode																
RND Da, Dn	1	1	1	<div style="display: flex; justify-content: space-between; align-items: center;"> 15 8 7 0 </div> <table border="1" style="width: 100%; text-align: center; border-collapse: collapse;"> <tr> <td style="width: 10%;">0</td> <td style="width: 10%;">*</td> <td style="width: 10%;">1</td> <td style="width: 10%;">1</td> <td style="width: 10%;">0</td> <td style="width: 10%;">1</td> <td style="width: 10%;">F</td> <td style="width: 10%;">F</td> <td style="width: 10%;">F</td> <td style="width: 10%;">1</td> <td style="width: 10%;">0</td> <td style="width: 10%;">0</td> <td style="width: 10%;">1</td> <td style="width: 10%;">J</td> <td style="width: 10%;">J</td> <td style="width: 10%;">J</td> </tr> </table>	0	*	1	1	0	1	F	F	F	1	0	0	1	J	J	J
0	*	1	1	0	1	F	F	F	1	0	0	1	J	J	J					

Note: ** indicates serial grouping encoding.

Instruction Fields

Dn **FFF** **Single Source/Destination Data Register**

000	D0	010	D2	100	D4	110	D6
001	D1	011	D3	101	D5	111	D7

Note: This instruction can specify D8-D15 as operands by using a prefix.

Da **JJJ** **Single Source Data Register**

000	D0	010	D2	100	D4	110	D6
001	D1	011	D3	101	D5	111	D7

Note: This instruction can specify D8-D15 as operands by using a prefix.

ROL Rotate One Bit Left Through the Carry Bit (DALU) ROL

Operation

$(Dn[38:0] \ll 1) \rightarrow Dn[39:1]$

$Dn[39] \rightarrow C$

$C \rightarrow Dn[0]$

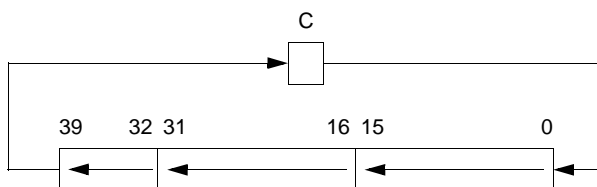
Assembler Syntax

ROL Dn

Description

ROL Dn

Rotates the contents of a data register (Dn) one bit to the left. The carry bit C is shifted to bit 0, bit 39 is copied to the carry bit, and bits [38:0] are copied to bits [39:1].



Status and Conditions that Affect Instruction

Register Address	Bit Name	Description
SR[0]	C	The carry bit is copied into Dn[0].

Status and Conditions Changed by Instruction

Register Address	Bit Name	Description
SR[0]	C	Set if bit 39 in the data register was one before rotation. Cleared if bit 39 in the data register was zero before rotation.
Ln	L	Clears the Ln bit in the destination register.

Example

```
rol d5
```

Register/Memory Address	Before	After
SR	\$00E4 0000	\$00E4 0001
L5:D5	\$0:\$FF A000 0005	\$0:\$FF 4000 000A



Instruction Formats and Opcodes

Instruction	Words	Cycles	Type	Opcode																
ROL Dn	1	1	1	<div style="display: flex; justify-content: space-between; align-items: center;"> 15 8 7 0 </div> <table border="1" style="width: 100%; text-align: center;"> <tr> <td>0</td><td>*</td><td>1</td><td>0</td><td>0</td><td>1</td><td>F</td><td>F</td><td>F</td><td>1</td><td>1</td><td>0</td><td>0</td><td>0</td><td>1</td><td>0</td> </tr> </table>	0	*	1	0	0	1	F	F	F	1	1	0	0	0	1	0
0	*	1	0	0	1	F	F	F	1	1	0	0	0	1	0					

Note: ** indicates serial grouping encoding.

Instruction Fields

Dn	FFF	Single Source/Destination Data Register						
	000	D0	010	D2	100	D4	110	D6
	001	D1	011	D3	101	D5	111	D7

Note: This instruction can specify D8-D15 as operands by using a prefix.

ROR Rotate One Bit Right Through the Carry Bit (DALU) ROR

Operation

$(Dn[39-1] \ggg 1) \rightarrow Dn[38-0]$

$C \rightarrow Dn[39]$

$Dn[0] \rightarrow C$

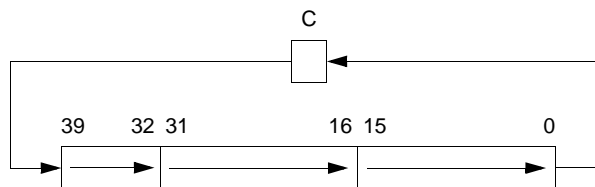
Assembler Syntax

ROR Dn

Description

ROR Dn

Rotates the contents of a data register (Dn) one bit to the right. The carry bit C is shifted to bit 39, bit 0 is copied to the carry bit, and bits [39:1] are copied to bits [38:0].



Status and Conditions that Affect Instruction

Register Address	Bit Name	Description
SR[0]	C	The carry bit is copied into Dn[39].

Status and Conditions Changed by Instruction

Register Address	Bit Name	Description
SR[0]	C	Set if bit 0 in the data register was one before rotation. Cleared if bit 0 in the data register was zero before rotation.
Ln	L	Clears the Ln bit in the destination register.

Example

ror d15

Register/Memory Address	Before	After
SR	\$00E0 0000	\$00E0 0001
L15:D15	\$0:\$FF A000 0005	\$0:\$7F D000 0002



Instruction Formats and Opcodes

Instruction	Words	Cycles	Type	Opcode																
ROR Dn	1	1	1	<div style="display: flex; justify-content: space-between; align-items: center;"> 15 8 7 0 </div> <table border="1" style="width: 100%; text-align: center;"> <tr> <td>0</td><td>*</td><td>1</td><td>0</td><td>0</td><td>1</td><td>F</td><td>F</td><td>F</td><td>1</td><td>1</td><td>0</td><td>0</td><td>0</td><td>1</td><td>1</td> </tr> </table>	0	*	1	0	0	1	F	F	F	1	1	0	0	0	1	1
0	*	1	0	0	1	F	F	F	1	1	0	0	0	1	1					

Note: ** indicates serial grouping encoding.

Instruction Fields

Dn	FFF	Single Source/Destination Data Register						
	000	D0	010	D2	100	D4	110	D6
	001	D1	011	D3	101	D5	111	D7

Note: This instruction can specify D8-D15 as operands by using a prefix.

Operation

(SP - 8) → PC
 (SP - 4) → SR
 SP - 8 → SP
 0 → NMID

Assembler Syntax

RTE

Description

RTE

Returns from an exception routine. The program counter and status register are popped from the active stack in memory, and program execution continues at the address specified in the PC. This instruction cannot appear in an execution set with another AGU instruction or a set that uses IFT and IFF, IFT and IFA, or IFF and IFA because RTE uses both AGUs. RTE does two simultaneous 32-bit long-word memory accesses. Instructions that change SR cannot appear in the same set with this instruction.

Note: Because RTE does not use RAS, returning from a subroutine using RTE is illegal. The result is undefined.

Status and Conditions that Affect Instruction

Register Address	Bit Name	Description
SR[18]	EXP	Determines which stack pointer used and execution working mode.

Status and Conditions Changed by Instruction

Register Address	Bit Name	Description
SR[31:0]		Restores SR from stack.
EMR[3]	NMID	Enables NMI.



xample

rte

Register/Memory Address	Before	After
ESP	\$00000010	\$00000008
(\$000C)	\$00E00000	
(\$0008)	\$0000000A	
PC		\$0000000A
SR	\$00E40000	\$00E00000
EMR		\$00000000

Instruction Formats and Opcodes

Instruction	Words	Cycles ¹	Type	Opcode
RTE	1	5/6	4	<div style="display: flex; justify-content: space-between; align-items: center;"> 15 8 7 0 </div> <div style="border: 1px solid black; padding: 2px; display: flex; justify-content: space-between;"> 1 0 0 1 1 1 1 1 0 1 1 1 0 0 1 1 </div>

Note 1: The shadow SP is valid or not valid. RTE uses 5 cycles if the shadow SP is valid. RTE uses 6 cycles if the shadow SP is not valid.

RTED Return From Exception With a Delay Slot (AGU) RTED

Operation

$(SP - 8) \rightarrow PC$
 $(SP - 4) \rightarrow SR$
 $SP - 8 \rightarrow SP$
 $0 \rightarrow NMID$

Assembler Syntax

RTED

Description

RTED

Returns from an exception routine after executing the execution set in the delay slot. The program counter and status register are popped from the active stack in memory, and program execution continues at the address specified in PC. This instruction cannot appear in an execution set with another AGU instruction or a set that uses IFT and IFF, IFT and IFA, or IFF and IFA because RTED uses both AGUs. RTED does two simultaneous 32-bit long-word memory accesses. Instructions that change SR cannot appear in the same set with this instruction or in the delay slot following the instruction.

Note: Because RTED does not use RAS, returning from a subroutine using RTED is illegal. The result is undefined.

Status and Conditions that Affect Instruction

Register Address	Bit Name	Description
SR[18]	EXP	Determines which stack pointer used and execution working mode.

Status and Conditions Changed by Instruction

Register Address	Bit Name	Description
SR[31:0]		Restores SR from stack.
EMR[3]	NMID	Enables NMI.



Example

rted

Instruction	Comment
move.w #\$2000,vba	Load the vector base address register.
trap	Issue a software interrupt and enter the exception state.
- - -	Instructions in the trap routine located at the address found at \$2000 and trap_vector offset.
rted not d4,d2	Execute the not instruction and the inc d1 instruction in the delay slot. Return to the original working mode (see example for RTE).
inc d1	

Instruction Formats and Opcodes

Instruction	Words	Cycles ¹	Type	Opcode								
RTED	1	5/6	4	<table border="1" style="margin-left: auto; margin-right: auto;"> <tr> <td style="text-align: center;">15</td> <td style="text-align: center;">8</td> <td style="text-align: center;">7</td> <td style="text-align: center;">0</td> </tr> <tr> <td style="text-align: center;">1 0 0 1 1 1 1 1</td> <td style="text-align: center;">0 1 1 1 0 0 1 0</td> <td></td> <td></td> </tr> </table>	15	8	7	0	1 0 0 1 1 1 1 1	0 1 1 1 0 0 1 0		
15	8	7	0									
1 0 0 1 1 1 1 1	0 1 1 1 0 0 1 0											

Note 1: The shadow SP is valid or not valid. RTED uses 5 cycles if the shadow SP is valid. RTED uses 6 cycles if the shadow SP is not valid. To get the correct cycle count for this instruction, subtract the execution time used by the execution set in the delay slot. The cycle count for this instruction cannot be less than 2 cycles.

Operation

Assembler Syntax

If (RAS valid), then RAS → PC;
 else (SP – 8) → PC;

RTS

always SP – 8 → SP

Description

RTS

Returns from a subroutine. If the RAS is valid, the PC is restored from the RAS. Otherwise, the PC is popped from the active stack in memory as a 32-bit long word. The stack pointer always decrements by 8, RAS becomes invalid, and program execution continues at the address specified in the PC.

Note: Because RTS uses the RAS mechanism, returning from an exception using RTS is illegal. The result is undefined.

Status and Conditions that Affect Instruction

Register Address	Bit Name	Description
SR[18]	EXP	Determines which stack pointer is used.

Status and Conditions Changed by Instruction

None.

Example

rts

Register/Memory Address	Before	After
SR	\$00E4 0000	
ESP	\$0000 2008	\$0000 2000
(\$2000)	\$0000 0018	
RAS	\$0000 0018	
PC	\$0000 0026	\$0000 0018



Instruction Formats and Opcodes

Instruction	Words	Cycles ¹	Type	Opcode
RTS	1	3/5/6	4	<div style="display: flex; justify-content: space-between; align-items: center;"> 15 8 7 0 </div> <div style="border: 1px solid black; padding: 2px; display: flex; justify-content: space-between; align-items: center;"> 1 0 0 1 1 1 1 1 0 1 1 1 0 0 0 1 </div>

Note 1: RTS uses 3 cycles if the RAS is valid. RTS uses 5 cycles if the RAS is not valid and the shadow SP is valid. RTS uses 6 cycles if neither the RAS nor the shadow SP are valid.

RTSD Return From Subroutine With Delay Slot (AGU) RTSD

Operation

If (RAS valid), then RAS \rightarrow PC;
 else (SP - 8) \rightarrow PC;

always SP - 8 \rightarrow SP

Assembler Syntax

RTSD

Description

RTSD

Returns from a subroutine after executing the execution set in the delay slot. If the RAS is valid, the PC is restored from the RAS. Otherwise, the PC is popped from the active stack in memory as a 32-bit long word. The implicit pop is done before the execution set in the delay slot is executed. The stack pointer always decrements by 8, RAS becomes invalid, and program execution continues at the address specified in the PC.

Note: Because RTSD uses the RAS mechanism, returning from an exception using RTSD is illegal. The result is undefined.

Status and Conditions that Affect Instruction

Register Address	Bit Name	Description
SR[18]	EXP	Determines which stack pointer is used.

Status and Conditions Changed by Instruction

None.

Example

rtsd

Instruction	Comment
	Initially ESP = \$2000
jsr SUB	Jump to subroutine at SUB.
- - -	Skip over these instructions.
SUB MOVE.w #\$20,d1	Execute the subroutine here. PC and SR pushed onto the stack at \$2000 and \$2004.
- - -	
rtsd move.w #\$47,d9 inc d9	Execute the \$47 to d9 and increment d9 to \$48, the instruction in the delay slot. Return from the subroutine. PC and SR popped from the stack.



Instruction Formats and Opcodes

Instruction	Words	Cycles ¹	Type	Opcode
RTSD	1	3/5/6	4	<div style="display: flex; justify-content: space-between; align-items: center;"> 15 8 7 0 </div> <div style="border: 1px solid black; padding: 2px; display: flex; justify-content: space-between;"> 1 0 0 1 1 1 1 1 0 1 1 1 0 0 0 0 </div>

Note 1: RTSD uses 3 cycles if the RAS is valid. RTSD uses 5 cycles if the RAS is not valid and the shadow SP is valid. RTSD uses 6 cycles if neither the RAS nor the shadow SP are valid. To get the correct cycle count for this instruction, subtract the execution time taken by the execution set in the delay slot. The cycle count for this instruction cannot be less than 1 cycle (2 cycles if shadow SP is not valid).

**Operation**

$(SP - 8) \rightarrow PC$
 $SP - 8 \rightarrow SP$

Assembler Syntax

RTSTK

Description**RTSTK**

Forces a return from a subroutine or exception by restoring the program counter (PC) from the active stack in memory. The restore to the PC is not from the RAS register, even if RAS is valid. The implicit pop is done before the execution set in the delay slot is executed. The stack pointer decrements by 8 and RAS becomes invalid. This instruction can be used to bypass RAS (for example, when the return address is changed directly on the stack). RTSTK does one 32-bit long-word memory access.

Status and Conditions that Affect Instruction

Register Address	Bit Name	Description
SR[18]	EXP	Determines which stack pointer is used.

Status and Conditions Changed by Instruction

Register Address	Bit Name	Description
EMR[3]	NMID	Cleared .



xample

rtstk

Instruction	Comment
- - -	
jsr SUB	Jump to subroutine at SUB. Push the PC and SR onto the stack.
- - -	Skip over these instructions.
SUB MOVE.w #\$16,d4	Execute the subroutine here.
- - -	
move.w #lbl,(SP-8)	Change the original value in the stack for PC to lbl.
rtstk	Restore the new value lbl to PC.
move.l #\$16,d5	This instruction skipped.
lbl move.l #\$16,d6	Continue executing here.

Instruction Formats and Opcodes

Instruction	Words	Cycles ¹	Type	Opcode
RTSTK	1	5/6	4	<div style="display: flex; justify-content: space-between; align-items: center;"> 15 8 7 0 </div> <div style="border: 1px solid black; padding: 2px; display: flex; justify-content: space-between;"> 1 0 0 1 1 1 1 1 0 1 1 1 0 1 0 1 </div>

Note 1: RTSTK uses 5 cycles if the shadow SP is valid. RTSTK uses 6 cycles if the shadow SP is not valid.

**Operation**

(SP - 8) → PC
 SP - 8 → SP

Assembler Syntax

RTSTKD

Description**RTSTKD**

Forces a return from a subroutine or exception by restoring the program counter (PC) from the active stack in memory after executing the execution set in the delay slot. The restore to the PC is not from the RAS register, even if RAS is valid. The implicit pop is done before the execution set in the delay slot is executed. The stack pointer decrements by 8 and RAS becomes invalid. This instruction can be used to bypass RAS (for example, when the return address is changed directly on the stack). RTSTK does one 32-bit long-word memory access.

Status and Conditions that Affect Instruction

Register Address	Bit Name	Description
SR[18]	EXP	Determines which stack pointer is used.

Status and Conditions Changed by Instruction

Register Address	Bit Name	Description
EMR[3]	NMID	Cleared .



xample

rtstkd

Instruction	Comment
- - -	
jsr SUB	Jump to subroutine at SUB. Push the PC and SR onto the stack.
- - -	Skip over these instructions.
SUB MOVE.w #\$16,d4	Execute the subroutine here.
- - -	
move.w #lbl,(SP-8)	Change the original value in the stack for PC to lbl.
rtstk move.l #\$35,d1 inc d1	Restore the new value lbl to PC. Load \$35 into d1. Increment d1 to \$36, the delay slot instruction.
move.l #\$16,d5	This instruction skipped.
lbl move.l #\$16,d6	Continue executing here.

Instruction Formats and Opcodes

Instruction	Words	Cycles ¹	Type	Opcode
RTSTKD	1	5/6	4	<div style="display: flex; justify-content: space-between; width: 100%;"> 15 8 7 0 </div> <div style="border: 1px solid black; padding: 2px; display: flex; justify-content: space-between;"> 1 0 0 1 1 1 1 1 0 1 1 1 0 1 0 0 </div>

Note 1: RTSTKD uses 5 cycles if shadow SP is valid. RTSTKD uses 6 cycles if the shadow SP is not valid. To get the correct cycle count for this instruction, subtract the execution time used by the execution set in the delay slot. The cycle count for this instruction cannot be less than 2 cycles.

Operation

If $D_a > \$007FFFFFFF$ then $\$007FFF0000 \rightarrow D_n$
 If $D_a < \$FF80000000$ then $\$FF80000000 \rightarrow D_n$
 Else $D_a \& \$FFFFFF0000 \rightarrow D_n$

Assembler Syntax

`SAT.F Da, Dn`

Description

SAT.F Da,Dn

If the values of the extension bits [39:32] and bit 31 of the source register are all zeros or all ones (no overflow), the source register is transferred to the destination register, and the LP is cleared. If the source register indicates an overflow, the saturated value (positive or negative depending on bit 39) is transferred to the HP of the destination register, sign-extended, and the LP is cleared. The saturated positive value is $\$007FFF0000$; the saturated negative value is $\$FF80000000$. This operation is independent of the SM bit in SR. It is intended for use after an instruction that is not affected by the saturation mode and before a MOVES instruction.

Status and Conditions that Affect Instruction

None.

Status and Conditions Changed by Instruction

Register Address	Bit Name	Description
Ln	L	Clears the Ln bit in the destination register.
EMR[2]	DOVF	Set if saturation occurs.

Example

`sat.f d2,d3`

Register/Memory Address	Before	After
L2:D2	$\$1 : \$00\ 846D\ 0000$	
L3:D3		$\$0 : \$00\ 7FFF\ 0000$
EMR		$\$0000\ 0004$



Instruction Formats and Opcodes

Instruction	Words	Cycles	Type	Opcode			
SAT.F Da, Dn	1	1	1	<div style="display: flex; justify-content: space-between; align-items: center;"> 15 8 7 0 </div> <table border="1" style="width: 100%; text-align: center;"> <tr> <td style="width: 33.33%;">0 * 1 1 0 1 F F</td> <td style="width: 33.33%;">F 1 0 1 1 J J J</td> <td style="width: 33.33%;"></td> </tr> </table>	0 * 1 1 0 1 F F	F 1 0 1 1 J J J	
0 * 1 1 0 1 F F	F 1 0 1 1 J J J						

Note: ** indicates serial grouping encoding.

Instruction Fields

Da **JJJ** **Single Source Data Register**

000	D0	010	D2	100	D4	110	D6
001	D1	011	D3	101	D5	111	D7

Note: This instruction can specify D8-D15 as operands by using a prefix.

Dn **FFF** **Single Source/Destination Data Register**

000	D0	010	D2	100	D4	110	D6
001	D1	011	D3	101	D5	111	D7

Note: This instruction can specify D8-D15 as operands by using a prefix.

Operation

If $D_n > \$007FFFFFFF$ then $\$007FFFFFFF \rightarrow D_n$ `SAT.L Dn`
 If $D_n < \$FF80000000$ then $\$FF80000000 \rightarrow D_n$
 Else $D_n \rightarrow D_n$

Assembler Syntax

Description

SAT.L Dn

If the values of the extension bits [39:32] and bit 31 of the source register are all zeros or all ones (no overflow), D_n is left alone. If the source register indicates an overflow, the saturated value (positive or negative depending on bit 39) is transferred to the destination register and sign-extended. The saturated positive value is $\$007FFFFFFF$; the saturated negative value is $\$FF80000000$. This operation is independent of the SM bit in SR. It is intended for use after an instruction that is not affected by the saturation mode and before a MOVES instruction.

Status and Conditions that Affect Instruction

None.

Status and Conditions Changed by Instruction

Register Address	Bit Name	Description
L_n	L	Clears the L_n bit in the destination register.
EMR[2]	DOVF	Set if saturation occurs.

Example

```
sat.l d6
```

Register/Memory Address	Before	After
L6:D6	\$1:\$00 828B 5E9E	\$0:\$00 7FFF FFFF
EMR		\$0000 0004

Instruction Formats and Opcodes

Instruction	Words	Cycles	Type	Opcode
SAT.L Dn	1	1	1	<div style="display: flex; justify-content: space-between; align-items: center;"> 15 8 7 0 </div> <div style="border: 1px solid black; padding: 2px; display: flex; justify-content: space-between; align-items: center;"> 0 * 1 0 0 1 F F F 1 1 0 0 1 0 1 </div>

Note: ** indicates serial grouping encoding.



Instruction Fields

Dn **FFF** **Single Source/Destination Data Register**

000	D0	010	D2	100	D4	110	D6
001	D1	011	D3	101	D5	111	D7

Note: This instruction can specify D8-D15 as operands by using a prefix.

Operation

$D_b - D_c - C \rightarrow D_d$

Assembler Syntax

SBC D_c, D_d

Description

SBC D_c, D_d

Subtracts the first data register (D_c) from the second (D_d), then subtracts the borrow (C bit) and stores the result in the second data register (D_d). The source operands are a data register pair. The destination register is the second register of the pair.

This instruction can be used in multiple-precision subtraction as illustrated in the example, which is a 64-bit subtraction.

Status and Conditions that Affect Instruction

Register Address	Bit Name	Description
SR[0]	C	Subtracted as a borrow from the LSB.
SR[5:4]	S[1:0]	Scaling mode bits determine which bits in the result are used in the L_n bit calculation.

Status and Conditions Changed by Instruction

Register Address	Bit Name	Description
EMR[2]	DOVF	Set if the result cannot be represented in 40 bits.
L_n	L	Calculates and updates the L_n bit in the destination register.
SR[0]	C	Calculates and updates the carry bit in the status register.

Note: The carry bit is set correctly for multiple-precision arithmetic using long word operands if the extension of the destination data register is the sign-extension of bit 31.

Example

```
sub d0,d1,d1
sbc d2,d3
```

Register/Memory Address	Before	After
D0	\$00 0000 0008	
L1:D1	\$0:\$00 0000 0005	\$0:\$FF FFFF FFFD
SR	\$00E4 0000	\$00E4 0001
D2	\$00 0000 0003	
L3:D3	\$0:\$00 0000 0005	\$0:\$00 0000 0001

Register/Memory Address	Before	After
SR	\$00E4 0001	\$00E4 0000
EMR		\$0000 0000

The two instructions shown can be used for a 64-bit subtraction, with the sub d0,d1,d1 performing the lower 32 bits, and the resultant borrow used for the LSB calculation of the upper 32 bits calculated by sbc d2,d3.

Instruction Formats and Opcodes

Instruction	Words	Cycles	Type	Opcode
SBC Dc, Dd	1	1	1	<div style="display: flex; justify-content: space-between; align-items: center;"> 15 8 7 0 </div> <div style="border: 1px solid black; padding: 2px; display: flex; justify-content: space-between; align-items: center;"> 0 * 1 0 1 1 e e 0 1 1 1 1 0 1 1 </div>

Note: ** indicates serial grouping encoding.

Instruction Fields

Dc, Dd	ee	Data Register Pairs					
00	D0, D1	01	D2, D3	10	D4, D5	11	D6, D7

Note: This instruction can specify D8-D15 as operands by using a prefix.

Operation

$Rnd(Dn - Da) \rightarrow Dn$

Assembler Syntax

`SBR Da, Dn`

Description

SBR Da, Dn

Subtracts the first data register (Da) of a pair from the second (Dn), then rounds the result and stores the result in the second data register (Dn). Rounding adjusts the LSB of the high part of the destination register according to the value of the low part of the register, and then zeros the low part. The two modes of the round function, Rnd(), are described on page A-359.

Status and Conditions that Affect Instruction

Register Address	Bit Name	Description
SR[2]	SM	If set, selects 32-bit arithmetic saturation mode.
SR[3]	RM	Rounding mode
SR[5:4]	S[1:0]	Scaling bits determine which bits in the result are used in the Ln bit calculation and which bits are used in rounding.

Status and Conditions Changed by Instruction

Register Address	Bit Name	Description
EMR[2]	DOVF	Set if the result cannot be represented in 40 bits, or if the result saturates to 32 bits in arithmetic saturation mode.
Ln	L	If not in arithmetic saturation mode (SR [SM] = 0), calculates and updates the Ln bit in the destination register. If in arithmetic saturation mode (SR [SM] = 1), clears the Ln bit in the destination register.

Example

`sbr d3, d0`

Register/Memory Address	Before	After
SR	\$00E0 0000	
D3	\$00 1539 0030	
L0:D0	\$0:\$00 2AE7 0080	\$0:\$00 15AE 0000
EMR		\$0000 0000

```

0010 1010 1110 0111 0000 0000 1000$2AE7 0080
- 0001 0101 0011 0000 0000 0000 0011$1539 0030
0001 0101 1010 1110 0000 0000 0101$15AE 0050
rounded                                $15AE 0000

```

Instruction Formats and Opcodes

Instruction	Words	Cycles	Type	Opcode																
SBR Da, Dn	1	1	1	<div style="display: flex; justify-content: space-between; align-items: center;"> 15 8 7 0 </div> <table border="1" style="width: 100%; text-align: center; border-collapse: collapse;"> <tr> <td style="width: 10%;">0</td> <td style="width: 10%;">*</td> <td style="width: 10%;">1</td> <td style="width: 10%;">1</td> <td style="width: 10%;">0</td> <td style="width: 10%;">0</td> <td style="width: 10%;">F</td> <td style="width: 10%;">F</td> <td style="width: 10%;">F</td> <td style="width: 10%;">1</td> <td style="width: 10%;">0</td> <td style="width: 10%;">0</td> <td style="width: 10%;">1</td> <td style="width: 10%;">J</td> <td style="width: 10%;">J</td> <td style="width: 10%;">J</td> </tr> </table>	0	*	1	1	0	0	F	F	F	1	0	0	1	J	J	J
0	*	1	1	0	0	F	F	F	1	0	0	1	J	J	J					

Note: ** indicates serial grouping encoding.

Instruction Fields

Da	JJJ	Single Source Data Register					
000	D0	010	D2	100	D4	110	D6
001	D1	011	D3	101	D5	111	D7

Note: This instruction can specify D8-D15 as operands by using a prefix.

Dn	FFF	Single Source/Destination Data Register					
000	D0	010	D2	100	D4	110	D6
001	D1	011	D3	101	D5	111	D7

Note: This instruction can specify D8-D15 as operands by using a prefix.



Operation

If $LC_n \leq 0$, then $PC + displacement \rightarrow PC$
 $0 \rightarrow LFn$

Assembler Syntax

SKIPLS label

Description

SKIPLS label

Branches to an address that is the current PC plus the displacement and disables the active loop if its loop counter (LCn) is less than or equal to zero. The displacement is calculated by the assembler and linker. SKIPLS is typically placed before a loop to bypass it if the loop count at run time does not indicate any iterations. Some programming rules apply to the use of this instruction. If no loops are enabled, this instruction is undefined.

Status and Conditions that Affect Instruction

Register Address	Bit Name	Description
SR[30:27]	LF[3:0]	Determines which loop is active.

Status and Conditions Changed by Instruction

Register Address	Bit Name	Description
SR[30:27]	LF[3:0]	Clear the active loop flag if the active loop counter is less than or equal to one.

Example

skipls _label

Register/Memory Address	Before	After
SR	\$10E0 0000	\$00E0 0000
_label (displacement)	\$0010	
PC	\$0000 000E	\$0000 001E



Instruction Formats and Opcodes

Instruction	Words	Cycles	Type	Opcode												
SKIPLS label	2	1/4 ¹	4	<table border="1"> <tr> <td style="text-align: center;">15</td> <td style="text-align: center;">8</td> <td style="text-align: center;">7</td> <td style="text-align: center;">0</td> </tr> <tr> <td>0 0 1 0 0 0 0 1</td> <td>A A A 0 0 0 1 1</td> <td></td> <td></td> </tr> <tr> <td>1 0 0 A A A A A</td> <td>A A A A A A A a</td> <td></td> <td></td> </tr> </table>	15	8	7	0	0 0 1 0 0 0 0 1	A A A 0 0 0 1 1			1 0 0 A A A A A	A A A A A A A a		
15	8	7	0													
0 0 1 0 0 0 0 1	A A A 0 0 0 1 1															
1 0 0 A A A A A	A A A A A A A a															

Note 1: If $LC > 1$, the instruction takes 1 cycle. If $LC \leq 0$ and the branch is taken, the instruction takes 4 cycles.

Instruction Fields

displacement	aAAAAAAAAAAAAAAAAA0	16-bit signed PC relative displacement. The encoding is the displacement with bit 0 stripped and replaced by the sign bit.
---------------------	---------------------	---

Operation

Enter the stop processing state.

Assembler Syntax

STOP

Description

STOP

Halts instruction execution and enters the STOP processing state. This state is intended for the lowest power consumption mode. The core informs the system about the intention to enter the STOP processing state, and it is up to the system to shut down the clocks.

All activity in the processor is halted until one of the following actions occurs:

- The wake_from_stop signal is asserted. In many chip configurations, this core interface signal is connected to one of the external interrupt request pins.
- A low level is applied to the RESET_B signal.
- A low level is applied to the EE0 debug signal.
- A JTAG debug request command is made.

Any of these actions causes the core to exit the STOP processing state, as follows:

If STOP is exited by assertion of the RESET signal, the processor enters the reset processing state.

If STOP is exited in parallel with an external interrupt request, the processor services the highest priority pending interrupt. If no interrupt is pending, or if no interrupt is enabled, the processor resumes execution at the instruction following the STOP instruction that caused entry into the stop state.

If STOP is exited by a low level on the EE0 signal or a JTAG debug request command, the processor enters the debug state immediately.

Status and Conditions that Affect Instruction

Register Address	Bit Name	Description
SR[18]	EXP	Determines execution working mode.

Status and Conditions Changed by Instruction

None

Instruction Formats and Opcodes

Instruction	Words	Cycles	Type	Opcode
STOP	1	8	4	<div style="display: flex; justify-content: space-between; align-items: center;"> 15 8 7 0 </div> <div style="border: 1px solid black; padding: 2px; display: flex; justify-content: space-between; align-items: center;"> 1 0 0 1 1 1 1 1 0 1 1 1 1 0 0 1 </div>



SUB

Subtract (DALU)

SUB

Operation

$D_n - \#u5 \rightarrow D_n$

$D_b - D_a \rightarrow D_n$

Assembler Syntax

`SUB #u5, Dn {0 ≤ u5 < 32}`

`SUB Da, Db, Dn`

Description

SUB #u5,Dn

Subtracts an immediate unsigned 5-bit value from a data register (Dn) and stores the result in the destination data register (Dn).

SUB Da,Db,Dn

Subtracts one source data register (Da) from a second data register (Db) and stores the result in a destination data register (Dn).

Status and Conditions that Affect Instruction

Register Address	Bit Name	Description
SR[2]	SM	If set, selects 32-bit arithmetic saturation mode.
SR[5:4]	S[1:0]	Scaling mode bits determine which bits in the result are used in the Ln bit calculation.

Status and Conditions Changed by Instruction

Register Address	Bit Name	Description
SR[0]	C	Calculates the borrow and updates the carry bit in the status register.
EMR[2]	DOVF	Set if the result cannot be represented in 40 bits, or if the result saturates to 32 bits in arithmetic saturation mode.
Ln	L	If not in arithmetic saturation mode (SR [SM] = 0), calculates and updates the Ln bit in the destination register. If in arithmetic saturation mode (SR [SM] = 1), clears the Ln bit in the destination register.

Example 1

```
sub d1, d0, d2
```

Register/Memory Address	Before	After
D0	\$00 0000 0005	
D1	\$00 0000 0008	
SR	\$00E4 0000	\$00E4 0001

Register/Memory Address	Before	After
L2:D2		\$0:\$FF FFFF FFFD
EMR		\$0000 0000

Example 2

sub d0,d1,d2

Register/Memory Address	Before	After
D0	\$FF D000 0000	
D1	\$00 2000 0000	
SR	\$00E4 0020	\$00E4 0021
L2:D2		\$1:\$00 5000 0000
EMR		\$0000 0000

Scaling up is set in SR[5], so L2 bit is set from overflow from bit 30.

Instruction Formats and Opcodes

Instruction	Words	Cycles	Type	Opcode								
SUB #u5,Dn	1	1	1	<table border="1"> <tr> <td>15</td> <td>8</td> <td>7</td> <td>0</td> </tr> <tr> <td>0 * 1 1 1 0 F F</td> <td>F 1 1 i i i i i</td> <td></td> <td></td> </tr> </table>	15	8	7	0	0 * 1 1 1 0 F F	F 1 1 i i i i i		
15	8	7	0									
0 * 1 1 1 0 F F	F 1 1 i i i i i											
SUB Da,Db,Dn	1	1	1	<table border="1"> <tr> <td>15</td> <td>8</td> <td>7</td> <td>0</td> </tr> <tr> <td>0 * 1 0 1 1 F F</td> <td>F 0 0 J J J J J</td> <td></td> <td></td> </tr> </table>	15	8	7	0	0 * 1 0 1 1 F F	F 0 0 J J J J J		
15	8	7	0									
0 * 1 0 1 1 F F	F 0 0 J J J J J											
SUB Db,Da,Dn	1	1	1	<table border="1"> <tr> <td>15</td> <td>8</td> <td>7</td> <td>0</td> </tr> <tr> <td>0 * 1 0 1 1 F F</td> <td>F 0 1 J J J J J</td> <td></td> <td></td> </tr> </table>	15	8	7	0	0 * 1 0 1 1 F F	F 0 1 J J J J J		
15	8	7	0									
0 * 1 0 1 1 F F	F 0 1 J J J J J											
SUB Da,Da,Dn	1	1	1	<table border="1"> <tr> <td>15</td> <td>8</td> <td>7</td> <td>0</td> </tr> <tr> <td>0 * 1 0 0 0 F F</td> <td>F 1 1 0 0 1 j j</td> <td></td> <td></td> </tr> </table>	15	8	7	0	0 * 1 0 0 0 F F	F 1 1 0 0 1 j j		
15	8	7	0									
0 * 1 0 0 0 F F	F 1 1 0 0 1 j j											

Note: ** indicates serial grouping encoding.

Operation

$Dn.H - Da.H \rightarrow Dn.H$
 $Dn.L - Da.L \rightarrow Dn.L$

Assembler Syntax

`SUB2 Da, Dn`

Description

SUB2 Da,Dn

Performs a 32-bit subtraction of source register Da from Dn with borrow disabled between bits 15 and 16 so that the high and low words of each register are subtracted separately. The result is stored back in Dn. The extension byte of the result is undefined.

Status and Conditions that Affect Instruction

None.

Status and Conditions Changed by Instruction

Register Address	Bit Name	Description
Ln	L	Clears the Ln bit in the destination register.

Example 1

`sub2 d0,d1`

Register/Memory Address	Before	After
D0	\$00 0003 2A14	
L1:D1	\$0:\$FF FFFE 2A18	\$0:\$FF FFFB 0004

Example 2

`sub2 d0,d1`

Register/Memory Address	Before	After
D0	\$00 7000 8000	
L1:D1	\$0:\$FF 8000 7000	\$0:\$FF 1000 F000



Instruction Formats and Opcodes

Instruction	Words	Cycles	Type	Opcode																
SUB2 Da, Dn	1	1	2	<div style="display: flex; justify-content: space-between; align-items: center;"> 15 8 7 0 </div> <table border="1" style="width: 100%; text-align: center;"> <tr> <td>1</td><td>1</td><td>0</td><td>1</td><td>0</td><td>0</td><td>F</td><td>F</td> <td>F</td><td>1</td><td>0</td><td>0</td><td>1</td><td>J</td><td>J</td><td>J</td> </tr> </table>	1	1	0	1	0	0	F	F	F	1	0	0	1	J	J	J
1	1	0	1	0	0	F	F	F	1	0	0	1	J	J	J					

Instruction Fields

Dn	FFF	Single Source/Destination Data Register						
	000	D0	010	D2	100	D4	110	D6
	001	D1	011	D3	101	D5	111	D7

Note: This instruction can specify D8-D15 as operands by using a prefix.

Da	JJJ	Single Source Data Register						
	000	D0	010	D2	100	D4	110	D6
	001	D1	011	D3	101	D5	111	D7

Note: This instruction can specify D8-D15 as operands by using a prefix.

Operation

$Rx - \#u5 \rightarrow Rx$

$Rx - rx \rightarrow Rx$

Assembler Syntax

`SUBA #u5,Rx {0 ≤ u5 < 64}`

`SUBA rx,Rx`

Description

This instruction subtracts an immediate or an AGU register from another AGU register. For R0-R7 destinations, this instruction is affected by the modifier mode selected in MCTL.

SUBA #u5,Rx

Subtracts an immediate unsigned 5-bit integer from an AGU register (Rx) and stores the result in the same register. If the stack pointer is the destination operand, then the immediate value must be a multiple of eight since the resulting 3 LSBs are forced to zero.

SUBA rx,Rx

Subtracts one AGU register (rx) from another (Rx) and stores the result in the destination AGU register (Rx). If the stack pointer is the destination operand, then the value in rx must be a multiple of eight since the resulting 3 LSBs are forced to zero.

Status and Conditions that Affect Instruction

Register Address	Bit Name	Description
SR[18]	EXP	Determines which stack pointer is used when the stack pointer is an operand. Otherwise, the instruction is not affected by SR.
MCTL[31:0]	AM3–AM0	Address modification bits when updating R0–R7. Otherwise, the instruction is not affected by MCTL.

Status and Conditions Changed by Instruction

None.

Example

```
suba r1,r0
```

Register/Memory Address	Before	After
MCTL	\$0000 0000	
R1	\$0000 0001	
R0	\$0000 0010	\$0000 000F

Operation

$$(2 * Dn) - Da \rightarrow Dn$$

Assembler Syntax

`SUBL Da, Dn`

Description

SUBL Da,Dn

Subtracts the source register (Da) from two times the destination register (Dn) and stores the result in the destination register. Dn is arithmetically shifted left one bit prior to the subtraction operation.

Status and Conditions that Affect Instruction

Register Address	Bit Name	Description
SR[2]	SM	If set, selects 32-bit arithmetic saturation mode.
SR[5:4]	S[1:0]	Scaling mode bits determine which bits in the result are used in the Ln bit calculation.

Status and Conditions Changed by Instruction

Register Address	Bit Name	Description
SR[0]	C	Calculates the borrow and updates the carry bit in the status register.
EMR[2]	DOVF	Set if the MS bit of the result cannot be represented in 40 bits, or saturates to 32 bits in arithmetic saturation mode, or the MS bit of the result changed due to the instruction's left shift operation.
Ln	L	If not in arithmetic saturation mode (SR [SM] = 0), calculates and updates the Ln bit in the destination register. If in arithmetic saturation mode (SR [SM] = 1), clears the Ln bit in the destination register.

Example 1

```
subl d0,d1
```

Register/Memory Address	Before	After
SR	\$00E0 0000	\$00E0 0000
D0	\$00 0000 0003	
L1:D1	\$0:\$00 0000 0004	\$0:\$00 0000 0005
EMR		\$0000 0000



Example 2

```
subl d0,d1
```

Register/Memory Address	Before	After
D0	\$00 0000 000A	
L1:D1	\$0:\$00 0000 0004	\$0:\$FF FFFF FFFE
SR	\$00E4 0000	\$00E4 0001

Instruction Formats and Opcodes

Instruction	Words	Cycles	Type	Opcode
SUBL Da,Dn	1	1	1	<div style="display: flex; justify-content: space-between; align-items: center;"> 15 8 7 0 </div> <div style="border: 1px solid black; padding: 2px; display: flex; justify-content: space-between; align-items: center;"> 0 * 1 1 0 0 F F F 1 0 1 1 J J J </div>

Note: ** indicates serial grouping encoding.

Instruction Fields

Da	JJJ	Single Source Data Register					
000	D0	010	D2	100	D4	110	D6
001	D1	011	D3	101	D5	111	D7

Note: This instruction can specify D8-D15 as operands by using a prefix.

Dn	FFF	Single Source/Destination Data Register					
000	D0	010	D2	100	D4	110	D6
001	D1	011	D3	101	D5	111	D7

Note: This instruction can specify D8-D15 as operands by using a prefix.



SUBNC.W

Subtract Without Changing the Carry Bit (DALU)

SUBNC.W

Operation

$Dn - \#s16 \rightarrow Dn$

Assembler Syntax

`SUBNC.W #s16,Dn {-215 ≤ s16 < 215}`

Description

`SUBNC.W #s16,Dn`

Subtracts an immediate signed 16-bit value from a source data register (Dn) and stores the result in the destination data register (Dn). The first operand is a 16-bit immediate data that is interpreted as a signed integer. The 16 bits are sign-extended to form a 32-bit operand. The carry bit is not affected by this instruction.

Status and Conditions that Affect Instruction

Register Address	Bit Name	Description
SR[2]	SM	If set, selects 32-bit arithmetic saturation mode.
SR[5:4]	S[1:0]	Scaling mode bits determine which bits in the result are used in the Ln bit calculation.

Status and Conditions Changed by Instruction

Register Address	Bit Name	Description
EMR[2]	DOVF	Set if the result cannot be represented in 40 bits, or if the result saturates to 32 bits in arithmetic saturation mode.
Ln	L	If not in arithmetic saturation mode (SR [SM] = 0), calculates and updates the Ln bit in the destination register. If in arithmetic saturation mode (SR [SM] = 1), clears the Ln bit in the destination register.

Example

`subnc.w #$15,d0`

Register/Memory Address	Before	After
SR	\$00E0 0000	\$00E0 0000
Immediate	\$0015	
L0:D0	\$0:\$00 0000 0010	\$0:\$FF FFFF FFFB
EMR		\$0000 0000



Instruction Formats and Opcodes

Instruction	Words	Cycles	Type	Opcode																																
SUBNC.W #s16,Dn	2	1	4	<div style="display: flex; justify-content: space-between; width: 100%;"> 15 8 7 0 </div> <table border="1" style="width: 100%; border-collapse: collapse;"> <tr> <td style="width: 8px;">0</td><td style="width: 8px;">0</td><td style="width: 8px;">1</td><td style="width: 8px;">1</td><td style="width: 8px;">1</td><td style="width: 8px;">1</td><td style="width: 8px;">0</td><td style="width: 8px;">0</td> <td style="width: 8px;">i</td><td style="width: 8px;">i</td><td style="width: 8px;">i</td><td style="width: 8px;">1</td><td style="width: 8px;">0</td><td style="width: 8px;">F</td><td style="width: 8px;">F</td><td style="width: 8px;">F</td> </tr> <tr> <td style="width: 8px;">1</td><td style="width: 8px;">0</td><td style="width: 8px;">0</td><td style="width: 8px;">i</td><td style="width: 8px;">i</td><td style="width: 8px;">i</td><td style="width: 8px;">i</td><td style="width: 8px;">i</td> <td style="width: 8px;">i</td><td style="width: 8px;">i</td><td style="width: 8px;">i</td><td style="width: 8px;">i</td><td style="width: 8px;">i</td><td style="width: 8px;">i</td><td style="width: 8px;">i</td><td style="width: 8px;">i</td> </tr> </table>	0	0	1	1	1	1	0	0	i	i	i	1	0	F	F	F	1	0	0	i	i	i	i	i	i	i	i	i	i	i	i	i
0	0	1	1	1	1	0	0	i	i	i	1	0	F	F	F																					
1	0	0	i	i	i	i	i	i	i	i	i	i	i	i	i																					

Instruction Fields

Dn	FFF	Single Source/Destination Data Register					
000	D0	010	D2	100	D4	110	D6
001	D1	011	D3	101	D5	111	D7

Note: This instruction can specify D8-D15 as operands by using a prefix.

#s16 iiiiiiiiiiiiiiii 16-bit signed immediate data

Operation

Assembler Syntax

Da[7:0] → Dn[7:0]; Da[7] → Dn[39:8]	SXT.B Da, Dn
Da[15:0] → Dn[15:0]; Da[15] → Dn[39:16]	SXT.W Da, Dn
Dn[31] → Dn[39:32]	SXT.L Dn

Description

These operations sign-extend a data register. The sign bit (bit 7 in a byte, bit 15 in a word, and bit 31 in a long word) is copied to the upper bits in a 40-bit data register.

SXT.B Da,Dn

Sign-extends a byte from a source data register (Da[7:0]) into a destination data register (Dn).

SXT.W Da,Dn

Sign-extends a word from a source data register (Da[15:0]) into a destination data register (Dn).

SXT.L Dn

Sign-extends a long word from a source data register (Dn[31:0]) into a destination data register (Dn).

Status and Conditions that Affect Instruction

None.

Status and Conditions Changed by Instruction

Register Address	Bit Name	Description
Ln	L	Clears the Ln bit in the destination register.

Example 1

```
sxt.b d3,d0
```

Register/Memory Address	Before	After
D3	\$FF FE34 A086	
L0:D0		\$0:\$FF FFFF FF86

Example 2

```
sxt.w d3,d2
```

Register/Memory Address	Before	After
D3	\$00 0000 7056	

Register/Memory Address	Before	After
L2:D2	\$0:00 B400 0000	\$0:\$00 0000 7056

Example 3

sxt.l d3

Register/Memory Address	Before	After
L3:D3	\$0:\$B4 8E60 6EC6	\$0:\$FF 8E60 6EC6

Instruction Formats and Opcodes

Instruction	Words	Cycles	Type	Opcode								
SXT.B Da, Dn	1	1	1	<table border="1"> <tr> <td>15</td> <td>8</td> <td>7</td> <td>0</td> </tr> <tr> <td>0 * 1 1 0 1 F F</td> <td>F 1 1 0 0 J J J</td> <td></td> <td></td> </tr> </table>	15	8	7	0	0 * 1 1 0 1 F F	F 1 1 0 0 J J J		
15	8	7	0									
0 * 1 1 0 1 F F	F 1 1 0 0 J J J											
SXT.W Da, Dn	1	1	1	<table border="1"> <tr> <td>15</td> <td>8</td> <td>7</td> <td>0</td> </tr> <tr> <td>0 * 1 1 0 1 F F</td> <td>F 1 1 1 0 J J J</td> <td></td> <td></td> </tr> </table>	15	8	7	0	0 * 1 1 0 1 F F	F 1 1 1 0 J J J		
15	8	7	0									
0 * 1 1 0 1 F F	F 1 1 1 0 J J J											
SXT.L Dn	1	1	1	<table border="1"> <tr> <td>15</td> <td>8</td> <td>7</td> <td>0</td> </tr> <tr> <td>0 * 1 0 0 1 F F</td> <td>F 1 1 0 0 0 0 1</td> <td></td> <td></td> </tr> </table>	15	8	7	0	0 * 1 0 0 1 F F	F 1 1 0 0 0 0 1		
15	8	7	0									
0 * 1 0 0 1 F F	F 1 1 0 0 0 0 1											

Note: ** indicates serial grouping encoding.

Instruction Fields

Da	JJJ	Single Source Data Register					
000	D0	010	D2	100	D4	110	D6
001	D1	011	D3	101	D5	111	D7

Note: This instruction can specify D8-D15 as operands by using a prefix.

Dn	FFF	Single Source/Destination Data Register					
000	D0	010	D2	100	D4	110	D6
001	D1	011	D3	101	D5	111	D7

Note: This instruction can specify D8-D15 as operands by using a prefix.

Operation

rx[7:0] → Rx[7:0]; rx[7] → Rx[31:8]
 Rx[15] → Rx[31:16]

Assembler Syntax

SXTA.B rx,Rx
 SXTA.W Rx

Description

These operations sign-extend an AGU register (address or offset register, program counter, or stack pointer). The sign bit (bit 7 in a byte or bit 15 in a word) is copied to the upper bits in a 32-bit AGU register.

SXTA.B rx,Rx

Sign-extends a byte from a source AGU register (rx[7:0]) into a destination AGU register (Rx).

SXTA.W Rx

Sign-extends a word from a source AGU register (Rx[15:0]) into a destination AGU register (Rx).

Status and Conditions that Affect Instruction

Register Address	Bit Name	Description
SR[18]	EXP	Determines which stack pointer is used when the stack pointer is an operand. Otherwise, the instruction is not affected by SR.

Status and Conditions Changed by Instruction

None.

Example 1

```
sxta.b r3,r1
```

Register/Memory Address	Before	After
R3	\$0000 2086	
R1		\$FFFF FF86

Example 2

```
sxta.w r3
```

Register/Memory Address	Before	After
R3	\$03BC 8A56	\$FFFF 8A56

Instruction Formats and Opcodes

Instruction	Words	Cycles	Type	Opcode																				
SXTA.B rx,Rx	1	1	2	<table border="1"> <tr> <td>15</td> <td>8</td> <td>7</td> <td>0</td> </tr> <tr> <td>1</td><td>1</td><td>1</td><td>0</td> <td>R</td><td>R</td><td>R</td><td>R</td> <td>1</td><td>1</td><td>0</td><td>0</td> <td>r</td><td>r</td><td>r</td><td>r</td> </tr> </table>	15	8	7	0	1	1	1	0	R	R	R	R	1	1	0	0	r	r	r	r
15	8	7	0																					
1	1	1	0	R	R	R	R	1	1	0	0	r	r	r	r									
SXTA.W Rx	1	1	2	<table border="1"> <tr> <td>15</td> <td>8</td> <td>7</td> <td>0</td> </tr> <tr> <td>1</td><td>1</td><td>1</td><td>0</td> <td>R</td><td>R</td><td>R</td><td>R</td> <td>1</td><td>1</td><td>1</td><td>1</td> <td>1</td><td>0</td><td>0</td><td>1</td> </tr> </table>	15	8	7	0	1	1	1	0	R	R	R	R	1	1	1	1	1	0	0	1
15	8	7	0																					
1	1	1	0	R	R	R	R	1	1	1	1	1	0	0	1									

Instruction Fields

rx	rrrr	AGU Source Register					
0000	N0	0100	—	1000	R0	1100	R4
0001	N1	0101	—	1001	R1	1101	R5
0010	N2	0110	PC	1010	R2	1110	R6
0011	N3	0111	SP	1011	R3	1111	R7

Note: This instruction can specify R8-R15 as operands by using a high register prefix.

Rx	RRRR	AGU Source/Destination Register					
0000	N0	0100	—	1000	R0	1100	R4
0001	N1	0101	—	1001	R1	1101	R5
0010	N2	0110	—	1010	R2	1110	R6
0011	N3	0111	SP	1011	R3	1111	R7

Note: This instruction can specify R8-R15 as operands by using a high register prefix.

Operation

Da → Dn

Assembler Syntax

TFR Da, Dn

Description

TFR Da, Dn

Copies a source data register (Da) to a destination data register (Dn). The Ln bit is re-calculated (not copied) in the destination register. Saturation mode is ignored and no saturation is done.

Status and Conditions that Affect Instruction

Register Address	Bit Name	Description
SR[5:4]	S[1:0]	Scaling bits determine which bits in the result are used in the Ln bit calculation.

Status and Conditions Changed By Instruction

Register Address	Bit Name	Description
Ln	L	Calculates and updates the Ln bit in the destination register.

Example 1

tfr d15, d14

Register/Memory Address	Before	After
SR	\$00E0 0000	
D15	\$FF F23A 1422	
L14:D14		\$0: \$FF F23A 1422

Example 2

tfr d7, d6

Register/Memory Address	Before	After
SR	\$00E0 0020	
D7	\$00 5000 0000	
D6		\$1: \$00 5000 0000

Scaling up set by SR[5], so L6 bit is set by bit 30 overflow.



Instruction Formats and Opcodes

Instruction	Words	Cycles	Type	Opcode																
TFR Da, Dn	1	1	1	<div style="display: flex; justify-content: space-between; align-items: center;"> 15 8 7 0 </div> <table border="1" style="width: 100%; text-align: center;"> <tr> <td>0</td><td>*</td><td>1</td><td>1</td><td>0</td><td>1</td><td>F</td><td>F</td><td>F</td><td>1</td><td>0</td><td>1</td><td>0</td><td>J</td><td>J</td><td>J</td> </tr> </table>	0	*	1	1	0	1	F	F	F	1	0	1	0	J	J	J
0	*	1	1	0	1	F	F	F	1	0	1	0	J	J	J					

Note: ** indicates serial grouping encoding.

Instruction Fields

Da	JJJ	Single Source Data Register						
	000	D0	010	D2	100	D4	110	D6
	001	D1	011	D3	101	D5	111	D7

Note: This instruction can specify D8-D15 as operands by using a prefix.

Dn	FFF	Single Source/Destination Data Register						
	000	D0	010	D2	100	D4	110	D6
	001	D1	011	D3	101	D5	111	D7

Note: This instruction can specify D8-D15 as operands by using a prefix.

Operation

$rx \rightarrow Rx$

Assembler Syntax

TFRA rx, Rx

Description

TFRA rx, Rx

Copies a source AGU register (rx) to a destination AGU register (Rx).

Status and Conditions that Affect Instruction

Register Address	Bit Name	Description
SR[18]	EXP	Determines which stack pointer is used when the stack pointer is an operand. Otherwise, the instruction is not affected by SR.

Status and Conditions Changed by Instruction

None.

Example

```
tfra r0,r1
```

Register/Memory Address	Before	After
R0	\$1234 5678	
R1		\$1234 5678

Instruction Formats and Opcodes

Instruction	Words	Cycles	Type	Opcode
TFRA rx, Rx	1	1	2	<div style="display: flex; justify-content: space-between; align-items: center;"> 15 8 7 0 </div> <div style="border: 1px solid black; padding: 2px; display: flex; justify-content: space-between; align-items: center;"> 1 1 1 0 R R R R 1 1 1 0 r r r r </div>

Instruction Fields

rx	rrrr	AGU Source Register					
0000	N0	0100	—	1000	R0	1100	R4
0001	N1	0101	—	1001	R1	1101	R5
0010	N2	0110	PC	1010	R2	1110	R6
0011	N3	0111	SP	1011	R3	1111	R7

Note: This instruction can specify R8-R15 as operands by using a high register prefix.

Operation

If (SR[EXP] = 1), then NSP → Rn
 else ESP → Rn

Assembler Syntax

TFRA OSP, Rn

If (SR[EXP] = 1), then Rn → NSP
 else Rn → ESP

TFRA Rn, OSP

Description

TFRA OSP, Rn

Writes the value of the inactive (other) stack pointer (OSP) to an address register (Rn). If EXP (SR[18]) is set, then OSP is the normal stack pointer (NSP). Otherwise, OSP is the exception stack pointer (ESP).

TFRA Rn, OSP

Writes the contents of an address register (Rn) to the inactive (other) stack pointer (OSP). If EXP (SR[18]) is set, then OSP is the normal stack pointer (NSP). Otherwise, OSP is the exception stack pointer (ESP).

Note: The value in NSP or ESP will have the lower three bits equal to zero.

Status and Conditions that Affect Instruction

Register Address	Bit Name	Description
SR[18]	EXP	Determines which stack pointer is used and execution working mode.

Status and Conditions Changed by Instruction

None

Operation

If T=1, then Da → Dn

If T=0, then Da → Dn

Assembler Syntax

TFRT Da, Dn

TFRF Da, Dn

Description

TFRT Da, Dn

Copies a source data register (Da) to a destination data register (Dn) if the T bit is set.

TFRF Da, Dn

Copies a source data register (Da) to a destination data register (Dn) if the T bit is cleared.

Status and Conditions that Affect Instruction

Register Address	Bit Name	Description
SR[1]	T	True bit
SR[5:4]	S[1:0]	Scaling bits determine which bits in the result are used in the Ln bit calculation.

Status and Conditions Changed by Instruction

Register Address	Bit Name	Description
Ln	L	Calculates and updates the Ln bit in the destination register.

Note: The Ln bit is re-calculated (not copied) in the destination register. Saturation mode is ignored and no saturation is done.

Example

tfrc d14,d15

Register/Memory Address	Before	After
SR	\$00E4 0002	
D14	\$FF F23A 1422	
L15:D15		\$0:\$FF F23A 1422



Instruction Formats and Opcodes

Instruction	Words	Cycles	Type	Opcode																
TFRT Da, Dn	1	1	2	<div style="display: flex; justify-content: space-between; margin-bottom: 5px;"> 15 8 7 0 </div> <table border="1" style="width: 100%; border-collapse: collapse;"> <tr> <td style="width: 8px;">1</td><td style="width: 8px;">1</td><td style="width: 8px;">0</td><td style="width: 8px;">1</td><td style="width: 8px;">0</td><td style="width: 8px;">0</td><td style="width: 8px;">F</td><td style="width: 8px;">F</td><td style="width: 8px;">F</td><td style="width: 8px;">1</td><td style="width: 8px;">0</td><td style="width: 8px;">1</td><td style="width: 8px;">0</td><td style="width: 8px;">J</td><td style="width: 8px;">J</td><td style="width: 8px;">J</td> </tr> </table>	1	1	0	1	0	0	F	F	F	1	0	1	0	J	J	J
1	1	0	1	0	0	F	F	F	1	0	1	0	J	J	J					
TFRF Da, Dn	1	1	2	<div style="display: flex; justify-content: space-between; margin-bottom: 5px;"> 15 8 7 0 </div> <table border="1" style="width: 100%; border-collapse: collapse;"> <tr> <td style="width: 8px;">1</td><td style="width: 8px;">1</td><td style="width: 8px;">0</td><td style="width: 8px;">1</td><td style="width: 8px;">0</td><td style="width: 8px;">0</td><td style="width: 8px;">F</td><td style="width: 8px;">F</td><td style="width: 8px;">F</td><td style="width: 8px;">1</td><td style="width: 8px;">0</td><td style="width: 8px;">1</td><td style="width: 8px;">1</td><td style="width: 8px;">J</td><td style="width: 8px;">J</td><td style="width: 8px;">J</td> </tr> </table>	1	1	0	1	0	0	F	F	F	1	0	1	1	J	J	J
1	1	0	1	0	0	F	F	F	1	0	1	1	J	J	J					

Instruction Fields

Da	JJJ	Single Source Data Register					
000	D0	010	D2	100	D4	110	D6
001	D1	011	D3	101	D5	111	D7

Note: This instruction can specify D8-D15 as operands by using a prefix.

Dn	FFF	Single Source/Destination Data Register					
000	D0	010	D2	100	D4	110	D6
001	D1	011	D3	101	D5	111	D7

Note: This instruction can specify D8-D15 as operands by using a prefix.

Operation

Next PC → (ESP), SR → (ESP + 4),
 ESP + 8 → ESP
 VBA[31:12]:trap_vector → PC
 0 → EXP
 0 → C
 0 → T
 00 → S[1:0]
 0 → SLF
 0000 → LS[3:0]

Assembler Syntax

TRAP {trap_vector = \$000}

Description

TRAPn

The TRAP instruction creates a **precise** software interrupt, halting execution and jumping to a code section pointed to from the exception table. The term **precise** is defined such that the exception timing is guaranteed to be synchronous with the instruction execution. The TRAP exception occurs immediately after the TRAP instruction. The current state of the machine is saved by pushing the values of the SR and the next PC onto the exception stack with two simultaneous 32-bit long-word memory accesses. The SR bits listed below are then set or cleared, including setting the interrupt priority level to the highest value (masking all maskable interrupts). The starting address of the exception processing routine is loaded to the PC and the Exception working mode is entered.

TRAP

The starting address of the exception processing routine is VBA[31:12]:\$000.

Status and Conditions that Affect Instruction

None.

Status and Conditions Changed by Instruction

Register Address	Bit Name	Description
SR[18]	EXP	Set
SR[0]	C	Cleared
SR[1]	T	Cleared
SR[5:4]	S[1:0]	Cleared
SR[31]	SLF	Cleared
SR[30:27]	LF[3:0]	Cleared
SR[23:21]	I[2:0]	Set interrupt priority level to 111.

Example 1

trap

Register/Memory Address	Before	After
ESP	\$0000 8030	\$0000 8038
VBA	\$8000 0000	
(\$8034)		\$00E0 0000
(\$8030)		\$0000 0014
PC	\$0000 0012	\$8000 0000
SR	\$00E0 0000	\$00E4 0000

Instruction Formats and Opcodes

Instruction	Words	Cycles	Type	Opcode
TRAP	1	5	4	<div style="display: flex; justify-content: space-between; align-items: center;"> 15 8 7 0 </div> <div style="border: 1px solid black; padding: 2px; display: flex; justify-content: space-between;"> 1 0 0 1 1 1 1 0 0 1 1 1 1 1 1 0 </div>

**Operation**If $D_n == 0$, then $1 \rightarrow T$, else $0 \rightarrow T$ **Assembler Syntax**TSTEQ D_n **Description**TSTEQ D_n Sets the T bit in SR if the source data register (D_n) is equal to zero; otherwise, it clears the T bit.**Status and Conditions that Affect Instruction**

None.

Status and Conditions Changed by Instruction

Register Address	Bit Name	Description
SR[1]	T	Set if the source operand is equal to zero and cleared if the source operand is not equal to zero.

Example

tsteq d1

Register/Memory Address	Before	After
D1	\$00 0000 0000	
SR	\$00E4 0000	\$00E4 0002

Instruction Formats and Opcodes

Instruction	Words	Cycles	Type	Opcode
TSTEQ D_n	1	1	1	<div style="display: flex; justify-content: space-between; width: 100%;"> 15 8 7 0 </div> <div style="border: 1px solid black; padding: 2px; display: flex; justify-content: space-between;"> 0 * 1 0 0 1 F F F 1 1 0 1 0 0 1 </div>

Note: ** indicates serial grouping encoding.**Instruction Fields**

Dn	Single Source/Destination Data Register							
	FFF							
	000	D0	010	D2	100	D4	110	D6
	001	D1	011	D3	101	D5	111	D7

Note: This instruction can specify D8-D15 as operands by using a prefix.**Note:****Note:**



TSTEQA.x Test for Equal to Zero (AGU) TSTEQA.x

Operation

If $Rx[15:0] == 0$, then $1 \rightarrow T$, else $0 \rightarrow T$

If $Rx[31:0] == 0$, then $1 \rightarrow T$, else $0 \rightarrow T$

Assembler Syntax

`TSTEQA.W Rx`

`TSTEQA.L Rx`

Description

Set the T bit if the source AGU register (Rx) is equal to zero; otherwise, clears the T bit.

TSTEQA.W Rx

Tests only the lower word (bits [15:0]) of the source operand.

TSTEQA.L Rx

Tests all 32 bits of the source operand.

Status and Conditions that Affect Instruction

Register Address	Bit Name	Description
SR[18]	EXP	Determines which stack pointer is used when the stack pointer is an operand. Otherwise, the instruction is not affected by SR.

Status and Conditions Changed by Instruction

Register Address	Bit Name	Description
SR[1]	T	Set if the source operand is equal to zero and cleared if the source operand is not equal to zero.

Example 1

```
tstega.w r4
```

Register/Memory Address	Before	After
R4	\$5F3E 0000	
SR	\$00E4 0000	\$00E4 0002

Example 2

```
tstega.l r1
```

Register/Memory Address	Before	After
R1	\$0000 0000	
SR	\$00E4 0000	\$00E4 0002

Instruction Formats and Opcodes

Instruction	Words	Cycles	Type	Opcode								
TSTEQA.W Rx	1	1	2	<table border="1"> <tr> <td style="text-align: right;">15</td> <td style="text-align: center;">8</td> <td style="text-align: center;">7</td> <td style="text-align: right;">0</td> </tr> <tr> <td>1 1 1 0 R R R R</td> <td>1 1 1 1</td> <td>0 0 0 0</td> <td></td> </tr> </table>	15	8	7	0	1 1 1 0 R R R R	1 1 1 1	0 0 0 0	
15	8	7	0									
1 1 1 0 R R R R	1 1 1 1	0 0 0 0										
TSTEQA.L Rx	1	1	2	<table border="1"> <tr> <td style="text-align: right;">15</td> <td style="text-align: center;">8</td> <td style="text-align: center;">7</td> <td style="text-align: right;">0</td> </tr> <tr> <td>1 1 1 0 R R R R</td> <td>1 1 1 1</td> <td>0 0 0 1</td> <td></td> </tr> </table>	15	8	7	0	1 1 1 0 R R R R	1 1 1 1	0 0 0 1	
15	8	7	0									
1 1 1 0 R R R R	1 1 1 1	0 0 0 1										

Instruction Fields

Rx	RRRR	AGU Source/Destination Register						
	0000	N0	0100	—	1000	R0	1100	R4
	0001	N1	0101	—	1001	R1	1101	R5
	0010	N2	0110	—	1010	R2	1110	R6
	0011	N3	0111	SP	1011	R3	1111	R7

Note: This instruction can specify R8-R15 as operands by using a high register prefix.



Operation

If $D_n \geq 0$, then $1 \rightarrow T$, else $0 \rightarrow T$

Assembler Syntax

TSTGE D_n

Description

TSTGE D_n

Sets the T bit if the source data register (D_n) is greater than or equal to zero; otherwise, clears the T bit. The value in D_n is treated as a signed number

Status and Conditions that Affect Instruction

None.

Status and Conditions Changed by Instruction

Register Address	Bit Name	Description
SR[1]	T	Set if the source operand is greater than or equal to zero and cleared if the source operand is not greater than or equal to zero.

Example

```
tstge d4
```

Register/Memory Address	Before	After
D4	\$00 5F3E 05C2	
SR	\$00E4 0000	\$00E4 0002

Instruction Formats and Opcodes

Instruction	Words	Cycles	Type	Opcode
TSTGE D_n	1	1	1	<div style="display: flex; justify-content: space-between; align-items: center;"> 15 8 7 0 </div> <div style="border: 1px solid black; padding: 2px; display: flex; align-items: center;"> 0 * 1 0 0 1 F F F 1 1 0 1 0 0 0 </div>

Note: ** indicates serial grouping encoding.

Instruction Fields

Dn	FFF						Single Source/Destination Data Register									
	000	D0	010	D2	100	D4	110	D6	001	D1	011	D3	101	D5	111	D7

Note: This instruction can specify D8-D15 as operands by using a prefix.



TESTGEA.L Test for Greater Than or Equal to Zero (AGU) TSTGEA.L

Operation

If $R_x \geq 0$, then $1 \rightarrow T$, else $0 \rightarrow T$

Assembler Syntax

TSTGEA.L Rx

Description

TESTGEA.L Rx

Sets the T bit if the source AGU register (Rx) is greater than or equal to zero; otherwise, it clears the T bit. The value in Rx is treated as a signed number.

Status and Conditions that Affect Instruction

Register Address	Bit Name	Description
SR[18]	EXP	Determines which stack pointer is used when the stack pointer is an operand. Otherwise, the instruction is not affected by SR.

Status and Conditions Changed by Instruction

Register Address	Bit Name	Description
SR[1]	T	Set if the source operand is greater than or equal to zero and cleared if the source operand is not greater than or equal to zero.

Example

```
tstgea.l r7
```

Register/Memory Address	Before	After
R7	\$57E3 A6CC	
SR	\$00E4 0000	\$00E4 0002



Instruction Formats and Opcodes

Instruction	Words	Cycles	Type	Opcode																
TSTGEA.L Rx	1	1	2	<div style="display: flex; justify-content: space-between; align-items: center;"> 15 8 7 0 </div> <table border="1" style="width: 100%; text-align: center;"> <tr> <td>1</td><td>1</td><td>1</td><td>0</td><td>R</td><td>R</td><td>R</td><td>R</td> <td>1</td><td>1</td><td>1</td><td>1</td><td>0</td><td>0</td><td>1</td><td>1</td> </tr> </table>	1	1	1	0	R	R	R	R	1	1	1	1	0	0	1	1
1	1	1	0	R	R	R	R	1	1	1	1	0	0	1	1					

Instruction Fields

Rx	RRRR	AGU Source/Destination Register						
	0000	N0	0100	—	1000	R0	1100	R4
	0001	N1	0101	—	1001	R1	1101	R5
	0010	N2	0110	—	1010	R2	1110	R6
	0011	N3	0111	SP	1011	R3	1111	R7

Note: This instruction can specify R8-R15 as operands by using a high register prefix.

Operation

If $D_n > 0$, then $1 \rightarrow T$, else $0 \rightarrow T$

Assembler Syntax

TSTGT Dn

Description

TSTGT Dn

Sets the T bit if the source data register (Dn) is greater than zero; otherwise, clears the T bit.

Status and Conditions that Affect Instruction

None.

Status and Conditions Changed by Instruction

Register Address	Bit Name	Description
SR[1]	T	Set if the source operand is greater than zero and cleared if the source operand is not greater than zero.

Example

tstgt d6

Register/Memory Address	Before	After
L6:D6	\$1:\$80 0000 0000	
SR	\$00E4 0002	\$00E4 0000

Instruction Formats and Opcodes

Instruction	Words	Cycles	Type	Opcode
TSTGT Dn	1	1	1	<div style="display: flex; justify-content: space-between; align-items: center;"> 15 8 7 0 </div> <div style="border: 1px solid black; padding: 2px; display: flex; align-items: center;"> 0 * 1 0 0 1 F F F 1 1 0 1 0 1 0 </div>

Note: ** indicates serial grouping encoding.

Instruction Fields

Dn	Single Source/Destination Data Register							
	FFF		D0		D1		D7	
	000	D0	010	D2	100	D4	110	D6
	001	D1	011	D3	101	D5	111	D7

Note: This instruction can specify D8-D15 as operands by using a prefix.



TSTGTA

Test for Greater Than Zero (AGU)

TSTGTA

Operation

If $R_x > 0$, then $1 \rightarrow T$, else $0 \rightarrow T$

Assembler Syntax

TSTGTA Rx

Description

TSTGTA Rx

Sets the T bit if the source AGU register (Rx) is greater than zero; otherwise, clears the T bit.

Status and Conditions that Affect Instruction

Register Address	Bit Name	Description
SR[18]	EXP	Determines which stack pointer is used when the stack pointer is an operand. Otherwise, the instruction is not affected by SR.

Status and Conditions Changed by Instruction

Register Address	Bit Name	Description
SR[1]	T	Set if the source operand is greater than zero and cleared if the source operand is not greater than zero.

Example

```
tstgta r2
```

Register/Memory Address	Before	After
R2	\$46EA 2BE8	
SR	\$00E4 0000	\$00E4 0002

Instruction Formats and Opcodes

Instruction	Words	Cycles	Type	Opcode
TSTGTA Rx	1	1	2	<div style="display: flex; align-items: center; gap: 10px;"> 15 8 7 0 </div> <div style="border: 1px solid black; padding: 2px; display: flex; align-items: center; gap: 2px;"> 1110RRRR11110010 </div>

Instruction Fields

Rx	RRRR	AGU Source/Destination Register					
0000	N0	0100	—	1000	R0	1100	R4
0001	N1	0101	—	1001	R1	1101	R5
0010	N2	0110	—	1010	R2	1110	R6
0011	N3	0111	SP	1011	R3	1111	R7

Note: This instruction can specify R8-R15 as operands by using a high register prefix.

Operation

Assembler Syntax

If VF2 == 1, then (D3.L << 1+1) → (word 3)*
 else (D1.L << 1+1) → (word 3)

VSL.4W D2:D6:D1:D3, (Rn)+N0

If VF0 == 1, then (D3.L << 1) → (word 2)
 else (D1.L << 1) → (word 2)
 D2.L → (word 0)
 D6.L → (word 1)

If VF3 == 1, then (D3.H << 1+1) → (word 3)
 else (D1.H << 1+1) → (word 3)

VSL.4F D2:D6:D1:D3, (Rn)+N0

If VF1 == 1, then (D3.H << 1) → (word 2)
 else (D1.H << 1) → (word 2)
 D2.H → (word 0)
 D6.H → (word 1)

If VF2 == 1, then (D3.L << 1+1) → (word 1)
 else (D1.L << 1+1) → (word 1)

VSL.2W D1:D3, (Rn)+N0

If VF0 == 1, then (D3.L << 1) → (word 0)
 else (D1.L << 1) → (word 0)

If VF3 == 1, then (D3.H << 1+1) → (word 1)
 else (D1.H << 1+1) → (word 1)

VSL.2F D1:D3, (Rn)+N0

If VF1 == 1, then (D3.H << 1) → (word 0)
 else (D1.H << 1) → (word 0)

Note: In the operation fields, the term << 1 indicates shift left 1 bit and fill the LSB with a zero. The term << 1+1 indicates shift left 1 bit and fill the LSB with a one.

For VSL.4W and VSL.4F, the high register quartet D10:D14:D9:D11 could be used with prefix encoding instead of D2:D6:D1:D3.

For VSL.2W and VSL.2F, the high register pair D9:D11 could be used with prefix encoding instead of D1:D3.

* Words 0, 1, 2, and 3 have different meanings in big and little endian modes, as follows:

Word	Memory Address	
	Big Endian Mode	Little Endian Mode
0	(Rn+2)	(Rn)
1	(Rn)	(Rn+2)
2	(Rn+6)	(Rn+4)
3	(Rn+4)	(Rn+6)



escription

The VSL instructions are intended to optimize the implementation of the Viterbi decoder algorithm. They are used in conjunction with the MAX2VIT instruction, which sets the Viterbi flags and stores the maximum portions of data register pairs into the destination registers for use with VSL. See MAX2VIT, page A-249.

The VSL instructions do not behave the same in little and big endian modes, meaning that data in source registers is written to different memory locations in the two modes. This behavior requires that the software implementation of Viterbi algorithms be different for the two endian modes. See [Section 2.4.1, “SC140 Endian Support,”](#) on page 2-56, for more detail on bus and memory behavior for each mode.

Note: The values in the data registers are not changed by these instructions.

VSL.4W D2:D6:D1:D3,(Rn)+N0

Writes four consecutive words taken from the LP of the source data registers to the memory. D2.L and D6.L are written to the location of the first two words in the memory, the order of which depends on the endian mode. The next two words written are: 1) A left-shifted value of D1.L or D3.L, according to the Viterbi flag VF0. If the Viterbi flag VF0 is set, then the left-shifted D3.L is chosen. Otherwise, the left-shifted D1.L is chosen and the LSB is filled with zero. 2) A left-shifted value of D1.L or D3.L, according to the Viterbi flag VF2. If the Viterbi flag VF2 is set, then the left-shifted D3.L is chosen. Otherwise, the left-shifted D1.L is chosen and the LSB is filled with one. The order of these two words depends on the endian mode. The address register values used with this instruction must be quad word-aligned (a multiple of 8).

VSL.4F D2:D6:D1:D3,(Rn)+N0

Writes four consecutive words taken from the HP of the source data registers to the memory. D2.H and D6.H are written to the location of the first two words in the memory, the order of which depends on the endian mode. The next two words that are written are: 1) A left-shifted value of D1.H or D3.H, according to the Viterbi flag VF1. If the Viterbi flag VF1 is set, then the left-shifted D3.H is chosen. Otherwise, the left-shifted D1.H is chosen and the LSB is filled with zero. 2) A left-shifted value of D1.H or D3.H, according to the Viterbi flag VF3. If the Viterbi flag VF3 is set, then the left-shifted D3.H is chosen. Otherwise, the left-shifted D1.H is chosen and the LSB is filled with one. The address register values used with this instruction must be quad word-aligned (a multiple of 8).

VSL.2W D1:D3,(Rn)+N0

Writes two consecutive words taken from the LP of the source data registers to the memory, the order of which depends on the endian mode. These words are: 1) A left-shifted value of D1.L or D3.L, according to the Viterbi flag VF0. If the Viterbi flag VF0 is set, then the left-shifted D3.L is chosen. Otherwise, the left-shifted D1.L is chosen and the LSB is filled with zero. 2) A left-shifted value of D1.L or D3.L, according to the Viterbi flag VF2. If the Viterbi flag VF2 is set, then the left-shifted D3.L is chosen. Otherwise, the left-shifted D1.L is chosen and the LSB is filled with one. The address register values used with this instruction must be long word-aligned (a multiple of 4).

VSL.2F D1:D3,(Rn)+N0

Writes two consecutive words taken from the HP of the source data registers to the memory, the order of which depends on the endian mode. These words are: 1) A left-shifted value of D1.H or D3.H, according to the Viterbi flag VF1. If the Viterbi flag VF1 is set, then the left-shifted D3.H is chosen. Otherwise, the left-shifted D1.H is chosen and the LSB is filled with zero. 2) A left-shifted value of D1.H or D3.H, according to the Viterbi flag VF3. If the Viterbi flag VF3 is set, then the left-shifted D3.H is chosen. Otherwise, the left-shifted D1.H is chosen and the LSB is filled with one. The address register values used with this instruction must be long word-aligned (a multiple of 4).



Status and Conditions that Affect Instruction

Register Address	Bit Name	Description
MCTL[31:0]	AM3–AM0	Address modification bits for R0–R7.
SR[8]	VF0	Viterbi flag 0 set by MAX2VIT D4,D2.
SR[9]	VF1	Viterbi flag 1 set by MAX2VIT D4,D2.
SR[10]	VF2	Viterbi flag 2 set by MAX2VIT D0,D6.
SR[11]	VF3	Viterbi flag 3 set by MAX2VIT D0,D6.
EMR[16]	BEM	Set if big endian mode, cleared if little endian mode.

Status and Conditions Changed by Instruction

None.

Example

vs1.2w d1:d3,(r0)+n0

Register/Memory Address	Before	After (Little Endian)	After (Big Endian)
MCTL	\$0000 0000		
SR	\$00e4 0000		
D1	\$00 2A62 EA79		
D3	\$00 5437 9EAC		
N0	\$0000 0002		
R0	\$0000 0060	\$0000 0068	\$0000 0068
\$0060		\$D4F2	\$D4F3
\$0062		\$D4F3	\$D4F2



Instruction Formats and Opcodes

Instruction	Words	Cycles	Type	Opcode								
VSL.4W D2:D6:D1:D3, (Rn)+N0	1	1	2	<table border="1"> <tr> <td>15</td> <td>8</td> <td>7</td> <td>0</td> </tr> <tr> <td>1 1 0 0 1 0 1 0</td> <td>0 0 0 0 0 R R R</td> <td></td> <td></td> </tr> </table>	15	8	7	0	1 1 0 0 1 0 1 0	0 0 0 0 0 R R R		
15	8	7	0									
1 1 0 0 1 0 1 0	0 0 0 0 0 R R R											
VSL.4F D2:D6:D1:D3, (Rn)+N0	1	1	2	<table border="1"> <tr> <td>15</td> <td>8</td> <td>7</td> <td>0</td> </tr> <tr> <td>1 1 0 0 1 0 1 0</td> <td>0 0 0 1 0 R R R</td> <td></td> <td></td> </tr> </table>	15	8	7	0	1 1 0 0 1 0 1 0	0 0 0 1 0 R R R		
15	8	7	0									
1 1 0 0 1 0 1 0	0 0 0 1 0 R R R											
VSL.2W D1:D3, (Rn)+N0	1	1	2	<table border="1"> <tr> <td>15</td> <td>8</td> <td>7</td> <td>0</td> </tr> <tr> <td>1 1 0 0 1 0 1 0</td> <td>0 0 1 0 0 R R R</td> <td></td> <td></td> </tr> </table>	15	8	7	0	1 1 0 0 1 0 1 0	0 0 1 0 0 R R R		
15	8	7	0									
1 1 0 0 1 0 1 0	0 0 1 0 0 R R R											
VSL.2F D1:D3, (Rn)+N0	1	1	2	<table border="1"> <tr> <td>15</td> <td>8</td> <td>7</td> <td>0</td> </tr> <tr> <td>1 1 0 0 1 0 1 0</td> <td>0 0 1 1 0 R R R</td> <td></td> <td></td> </tr> </table>	15	8	7	0	1 1 0 0 1 0 1 0	0 0 1 1 0 R R R		
15	8	7	0									
1 1 0 0 1 0 1 0	0 0 1 1 0 R R R											

Instruction Fields

Rn	RRR	Address Register					
	000 R0	010 R2	100 R4	110 R6			
	001 R1	011 R3	101 R5	111 R7			

Note: This instruction can specify R8-R15 as operands by using a high register prefix.

Operation

Enters the low-power standby WAIT processing state.

Assembler Syntax

WAIT

Description

WAIT

Enters the low-power standby WAIT processing state. All internal core processing is halted until an unmasked interrupt occurs, the DSP is reset, the EE0 is asserted, or a JTAG debug request command is issued. If an exit from the WAIT processing state is caused by asserting EE0 or the JTAG debug request command, the processor enters the debug state immediately.

The WAIT processing state is intended to be an intermediate power consumption mode between the execution processing state and the STOP processing state. The decision what parts of the system other than the core will have their clocks shut down in the WAIT processing state and what parts will continue to operate is system dependent. Common examples are peripherals that might receive data and memories that can be accessed by DMA controllers, which interrupt the core when data is available for processing.

The WAIT instruction can appear only once in an execution set.

During the WAIT processing state, if a maskable interrupt is asserted, the core behaves according to the following rules:

Condition

The priority level of the interrupt is higher than the level programmed in the SR by the IPLn bits, and the DI bit in SR is clear (meaning the interrupt is enabled).

The priority level of the interrupt is higher than the level programmed in the SR by the IPLn bits, and the DI bit in SR is set (meaning the interrupt is masked only by the DI bit).

The priority level of the interrupt is lower than or equal to the level programmed in the SR by the IPLn bits.

A non-maskable interrupt is asserted.

Response

Exit the WAIT state and service the interrupt immediately after the execution set that included the WAIT instruction.

Exit the WAIT state and continue execution of the execution set that included the WAIT instruction. Do not jump to the interrupt service routine.

Remain in the WAIT state.

Exit the WAIT state and service the non-maskable interrupt immediately after the execution set that included the WAIT instruction, regardless of the value of the IPL and DI bits in the SR.

Status and Conditions that Affect Instruction

Register Address	Bit Name	Description
SR[18]	EXP	Determines execution working mode.

Status and Conditions Changed by Instruction

None

Instruction Formats and Opcodes

Instruction	Words	Cycles	Type	Opcode
WAIT	1	8	4	<div style="display: flex; justify-content: space-between; align-items: center;"> 15 8 7 0 </div> <div style="border: 1px solid black; padding: 2px; display: flex; justify-content: space-between;"> 1 0 0 1 1 1 1 1 0 1 1 1 1 0 0 0 </div>

Operation

Da[7:0] → Dn[7:0]; 0 → Dn[39:8]

Da[15:0] → Dn[15:0]; 0 → Dn[39:16]

0 → Dn[39:32]

Assembler Syntax

ZXT.B Da, Dn

ZXT.W Da, Dn

ZXT.L Dn

Description

These operations zero-extend a data register.

ZXT.B Da,Dn

Copies bits [7:0] from a source data register (Da) to a 40-bit destination data register (Dn) and zero-extends bits [39:8] of Dn.

ZXT.W Da,Dn

Copies bits [15:0] from a source data register (Da) to a 40-bit destination data register (Dn) and zero-extends bits [39:16] of Dn.

ZXT.L Dn

Zero-extend a long word from bit 32 through the remaining upper bits in a 40-bit data register (Dn).

Status and Conditions that Affect Instruction

None.

Status and Conditions Changed by Instruction

Register Address	Bit Name	Description
Ln	L	Clears the Ln bit in the destination register.

Example 1

```
zxt.b d2,d5
```

Register/Memory Address	Before	After
D2	\$00 46EA 8BE8	
L5:D5		\$0:\$00 0000 00E8

Example 2

```
zxt.w d3,d6
```

Register/Memory Address	Before	After
D3	\$FF A836 5EC4	

Register/Memory Address	Before	After
L6:D6		\$0:\$00 0000 5EC4

Example 3

zxt.l d0

Register/Memory Address	Before	After
L0:D0	\$0:\$FF A836 A7C4	\$0:\$00 A836 A7C4

Instruction Formats and Opcodes

Instruction	Words	Cycles	Type	Opcode									
ZXT.B Da, Dn	1	1	1	<table border="1"> <tr> <td>15</td> <td>8</td> <td>7</td> <td>0</td> </tr> <tr> <td>0</td> <td>*</td> <td>1</td> <td>1 0 1 F F</td> <td>F 1 1 0 1 J J J</td> </tr> </table>	15	8	7	0	0	*	1	1 0 1 F F	F 1 1 0 1 J J J
15	8	7	0										
0	*	1	1 0 1 F F	F 1 1 0 1 J J J									
ZXT.W Da, Dn	1	1	1	<table border="1"> <tr> <td>15</td> <td>8</td> <td>7</td> <td>0</td> </tr> <tr> <td>0</td> <td>*</td> <td>1</td> <td>1 0 1 F F</td> <td>F 1 1 1 1 J J J</td> </tr> </table>	15	8	7	0	0	*	1	1 0 1 F F	F 1 1 1 1 J J J
15	8	7	0										
0	*	1	1 0 1 F F	F 1 1 1 1 J J J									
ZXT.L Dn	1	1	1	<table border="1"> <tr> <td>15</td> <td>8</td> <td>7</td> <td>0</td> </tr> <tr> <td>0</td> <td>*</td> <td>1</td> <td>0 0 1 F F</td> <td>F 1 1 0 0 0 0 0</td> </tr> </table>	15	8	7	0	0	*	1	0 0 1 F F	F 1 1 0 0 0 0 0
15	8	7	0										
0	*	1	0 0 1 F F	F 1 1 0 0 0 0 0									

Note: ** indicates serial grouping encoding.

Instruction Fields

Da	JJJ	Single Source Data Register					
000	D0	010	D2	100	D4	110	D6
001	D1	011	D3	101	D5	111	D7

Note: This instruction can specify D8-D15 as operands by using a prefix.

Dn	FFF	Single Source/Destination Data Register					
000	D0	010	D2	100	D4	110	D6
001	D1	011	D3	101	D5	111	D7

Note: This instruction can specify D8-D15 as operands by using a prefix.

Operation

$rx[7:0] \rightarrow Rx[7:0]; 0 \rightarrow Rx[31:8]$

$0 \rightarrow Rx[31:16]$

Assembler Syntax

`ZXTA.B rx,Rx`

`ZXTA.W Rx`

Description

These operations zero-extend an AGU source register (address or offset register, program counter, or stack pointer).

ZXTA.B rx,Rx

Copies bits [7:0] from a source AGU register (rx) to a 32-bit destination AGU register (Rx) and zero-extends bits [31:8] of Rx.

ZXTA.W Rx

Zero-extends bits [31:16] of Rx.

Status and Conditions that Affect Instruction

Register Address	Bit Name	Description
SR[18]	EXP	Determines which stack pointer is used when the stack pointer is an operand. Otherwise, the instruction is not affected by SR.

Status and Conditions Changed by Instruction

None.

Example 1

```
zxta.b r3,n2
```

Register/Memory Address	Before	After
R3	\$E4A6 5C8A	
N2		\$0000 008A

Example 2

```
zxta.w r4
```

Register/Memory Address	Before	After
R4	\$E4A6 5C8A	\$0000 5C8A

Instruction Formats and Opcodes

Instruction	Words	Cycles	Type	Opcode																				
ZXTA.B rx,Rx	1	1	2	<table border="1"> <tr> <td>15</td> <td>8</td> <td>7</td> <td>0</td> </tr> <tr> <td>1</td><td>1</td><td>1</td><td>0</td> <td>R</td><td>R</td><td>R</td><td>R</td> <td>1</td><td>1</td><td>0</td><td>1</td> <td>r</td><td>r</td><td>r</td><td>r</td> </tr> </table>	15	8	7	0	1	1	1	0	R	R	R	R	1	1	0	1	r	r	r	r
15	8	7	0																					
1	1	1	0	R	R	R	R	1	1	0	1	r	r	r	r									
ZXTA.W Rx	1	1	2	<table border="1"> <tr> <td>15</td> <td>8</td> <td>7</td> <td>0</td> </tr> <tr> <td>1</td><td>1</td><td>1</td><td>0</td> <td>R</td><td>R</td><td>R</td><td>R</td> <td>1</td><td>1</td><td>1</td><td>1</td> <td>1</td><td>0</td><td>0</td><td>0</td> </tr> </table>	15	8	7	0	1	1	1	0	R	R	R	R	1	1	1	1	1	0	0	0
15	8	7	0																					
1	1	1	0	R	R	R	R	1	1	1	1	1	0	0	0									

Instruction Fields

rx	rrrr	AGU Source Register					
0000	N0	0100	—	1000	R0	1100	R4
0001	N1	0101	—	1001	R1	1101	R5
0010	N2	0110	PC	1010	R2	1110	R6
0011	N3	0111	SP	1011	R3	1111	R7

Note: This instruction can specify R8-R15 as operands by using a high register prefix.

Rx	RRRR	AGU Source/Destination Register					
0000	N0	0100	—	1000	R0	1100	R4
0001	N1	0101	—	1001	R1	1101	R5
0010	N2	0110	—	1010	R2	1110	R6
0011	N3	0111	SP	1011	R3	1111	R7

Note: This instruction can specify R8-R15 as operands by using a high register prefix.



Appendix B

StarCore Registry

The StarCore registry (SCR) is a system that identifies the core version.

B.1 Using the StarCore Registry

The SCR is to be used for debugging software and run-time software. A StarCore Identification number, SCID, is encoded in bits 23-17 of the EOnCE Status Register (ESR). This 32-bit, memory-mapped, read-only register is located at offset 00 from the EOnCE register base address defined by each System-on-a-Chip (SoC) derivative. The SCID can be read through the JTAG standard test interface or run-time software using the MOVE instruction. The SCID binary value can be placed in a data register by reading ESR, shifting right by 17 bits, and ANDING the result with a \$0000 007F mask. Reading ESR using SC100 instructions does not activate the EOnCE block, thus saving power in actual operation.

An example code sequence is:

```

move.w ($00800002),d4      ;reads from memory location of high portion ESR, assuming
                           ;that the EOnCE base address is $0080 0000. Bits 31-16 of
                           ;the ESR go to bits 15-0 of D4.
lsr d4                    ;shifts right 1 bit
and #0,d4.h               ;clears high portion
and #$007f,d4.l          ;clears bits 15-7, leaves the SCID in data register d4

```

The SCID has three fields:

- REVNO (bits 23-21) instruction set version
- RESERVED (bit 20) - 0 in Freescale implementations
- CORETP (bits 19-17) core architecture version

The REVNO field generally identifies the basic instruction set revision of the SC100 core. It identifies the availability of new instructions and corrections to existing instructions. Binary-encoded programs will generally run without modification on later versions of the instruction set. Changes in REVNO imply a software tools switch, different software simulator and different host debugger.

The CORETP field identifies the architecture member within the SC100 family. It identifies the availability of new execution units and VLES grouping capabilities. Note that execution units and VLES can scale up or down without altering the basic instruction set. Changes in CORETP imply a software tools switch, different software simulator and different host debugger.

The following table lists current assignments of REVNO and CORETP. The Tools column lists the first version of StarCore software development tools to support the listed Instruction Set Version.



Table B-1. SCID Assignments

Hex SCID	Bits 23-21 REVNO	Bit 20 Reserved	Bits 19-17 CORETP	Instruction Set Version	Cores	Example SoC / platform
00	000	0	000	Original	SC140 rev 0	Pre-Chip
10	001	0	000	V1	SC140 rev 0_1	MSC8101 rev0
11	001	0	001	V1	SC140 rev 0_1	Rainbow
21	010	0	001	V2	SC140 rev A	MSC8102 MCS8101 revA
31	011	0	001	V3	SC140e	P2002

In SC100 implementations, the SCID is defined at the SoC level by strapping a set of core interface signals that define the REVNO and CORETP fields during reset. Therefore, all SoC designs must conform to the SCID to derive the core identification benefits for host-based software tools and run-time software.

The SCID should not be used for SoC or mask set identification outside the control of StarCore. SC100-based products should add an off-core SoC or mask set identification register, independent of the SCR. The SCR is used only for on-core identification.

The SCID may assist in migrating applications software and software development tools across StarCores. Software tools developers and run-time software need to assess the whole SCID (both REVNO, RESERVED and CORETP fields) to determine possible code migration from core to core. Note that the SCR does not guarantee identical timing across SCIDs.

For the same CORETP, software written for a lower REVNO will produce the same results on a higher REVNO except where the higher REVNO introduces a bug fix to an existing instruction. These bug fixes may present software migration and tools issues.

For the same REVNO, software written for a lower CORETP may not run on a higher CORETP because CORETP is not a monotonic scalability index. A higher CORETP may have more or less execution units and VLES grouping capabilities than a lower CORETP. For example, software written for the SC140 (CORETP = 001) will not run and produce the same results on a dual-MAC SC100, having the same or higher REVNO. Software tools developers and run-time software need to be aware of the actual CORETP software migration issues.

For historical reasons, the CORETP field of the Rainbow product is 000 instead of 001. However the SW tools should consider this product as identical with the MCS8101 rev0 product.

A

AAU (address arithmetic unit) 1-3, 2-4
 ABS A-20
 Accelerator 2-5, 6-57
 Access width support 2-42
 ADC A-22
 ADD A-24
 ADD2 A-27
 ADDA A-29
 ADDL1A A-32
 ADDL2A A-34
 ADDNC.W A-36
 Address 2-35
 Address generation pipeline stage 5-4
 Address modifier modes 2-45
 linear addressing mode 2-45
 modulo addressing mode 2-45
 multiple wrap-around modulo addressing mode 2-47
 Address register indirect modes
 address modifier modes 2-45
 Address registers 2-4, 2-35, 2-37
 Addressing modes
 PC relative mode 2-40
 register direct mode 2-38
 register indirect mode 2-38
 special address modes 2-41
 special mode 2-41
 summary 2-43
 ADR A-38
 AGU
 architecture 2-31
 arithmetic instructions 2-48
 block diagram 2-32
 programming model 2-34
 AGU (address generation unit) 1-3, 2-3, 2-31, 2-34, 2-64
 ALU (arithmetic logic unit) 1-3, 2-2
 AM (address modification bits) 2-37
 AM bits 2-45
 AND A-40, A-43
 AND.W A-45
 Arithmetic instructions on address registers 2-48
 Arithmetic saturation mode 2-25
 bit 3-6
 ASL A-48
 ASL2A A-50

ASLA A-51
 ASLL A-52
 ASLW A-55
 ASR A-57
 ASRA A-59
 ASRR A-60
 ASRW A-63
 ATS (access type selection) 4-56, 4-60
 AWS (access width selection) 4-59

B

B0-B7 (base address registers) 2-36
 BEM (big endian memory bit) 3-8
 BF A-65
 BFD A-67
 BFU (bit-field unit) 2-2, 2-3, 2-12
 Bit mask
 instructions 2-49
 semaphore support instructions 2-50
 Bit mask instructions 2-67
 BMCHG A-69
 BMCHG.W A-69
 BMCLR A-75
 BMCLR.W A-78
 BMSET A-80
 BMSET.W A-82
 BMTSET A-84
 BMTSET.W A-86
 BMTSTC A-89
 BMTSTC.W A-91
 BMTSTS A-94
 BMTSTS.W A-96
 BMU (bit mask unit) 1-3, 2-4
 BRA A-99
 BRAD A-101
 BREAK A-103
 BS (bus selection) 4-56
 BSR A-105
 BSRD A-107
 BT A-109
 BTD A-111

C

C (carry bit) 3-7
 CACS (comparator A condition selection) 4-56
 Carry bit 3-7
 CBCS (comparator B condition selection) 4-56



- CCS (comparator condition selection bits) 4-60
- Change-of-flow instructions 2-68
- CLB A-113
- CLR A-115
- CMPEQ A-117
- CMPEQ.W A-119
- CMPEQA A-121
- CMPGT A-123
- CMPGT.W A-125
- CMPGTA A-127
- CMPHI A-129
- CMPHIA A-131
- Conditional execution 5-9
- CONT A-133
- CONTD A-135
- Control instructions 2-68
- Control registers 3-1
- Convergent rounding 2-21
- Core Architecture 1-1
- Core control registers 3-1
 - clearing EMR bits 3-10
 - exception and mode register (EMR) 3-7
 - status register (SR) 3-1, 3-2
- CORES (core status) 4-38
- CORETP (core type) 4-39
- CORETP (SCID field) B-1
- CS (comparators selection) 4-55

D

- D0-D15 data registers 2-8
- data 2-1
- Data ALU (data arithmetic logic unit) 1-3, 2-2
 - architecture 2-6
 - arithmetic and rounding 2-17
 - arithmetic saturation mode 2-25
 - bit-field unit 2-3, 2-12
 - data formats 2-18
 - data shifter/limiter 2-13
 - multi-precision arithmetic support 2-26
 - programming model 2-7
 - rounding 2-21
 - scaling 2-14
 - signed fractional 2-18
 - signed integer 2-19
 - unsigned comparison 2-21
 - unsigned integer 2-19
- Data buses 2-2
- Data buses (XDBA and XDBB) 2-6, 2-9
- Data formats 2-18
- Data registers (D0-D15) 2-3, 2-8
 - accesses 2-8
- Data shifter/limiter 2-13
- DEBUG A-137
- Debug exception 4-12

- Debug mode 4-11
- DEBUGERST (debugger status information) 4-42
- DEBUGEV A-138
- Debugging system 4-1
- DECA A-139
- DECEQ A-141
- DECEQA A-143
- DECGE A-144
- DECGEA A-146
- DI A-148
- DI (disable interrupts bit) 3-4
- DIS (debug interrupt status) 4-42
- DIV A-150
- Division 2-20
- DMA (direct memory access) 1-4
- DMAC implementation 2-26
- DMACSS A-153
- DMACSU A-155
- DOENn A-157
- DOENSHn A-159
- DOSETUPn A-161
- DOVF (data ALU overflow bit) 3-8
- DRCOUNTER (debug reason is counter) 4-40
- DREDCA7-0 (debug reason is EDCA7-0) 4-40
- DREE4-0 (debug reason is EE4-0) 4-39
- DRSW (debug reason is software bug) 4-39
- DRTBFULL (debug reason is trace buffer) 4-39

E

- ECNT_CTRL (event counter control register) 4-50
- ECNT_CTRL register
 - ECNTEN 4-52
 - ECNTWHAT 4-52
 - EXT 4-51
- ECNT_EXT (extension counter value register) 4-53
- ECNT_VAL (event counter value register) 4-52
- ECNTEN (event counter enable) 4-52
- ECNTWHAT (events to be counted) 4-52
- ECR (EOnCE command register) 4-36
 - EX 4-37
 - GO 4-37
 - REGSEL 4-37
- EDCA (address event detection channel) 4-22, 4-54
 - control registers (EDCAi_CTRL) 4-54
 - mask registers (EDCAi_MASK) 4-57
 - reference value registers A and B (EDCAi_REFA, EDCAi_REFB) 4-57
- EDCAEN (event detection channel (EDCAi) enable) 4-55
- EDCAi_CTRL (EDCA control registers) 4-54
 - ATS 4-56
 - BS 4-56
 - CACS 4-56
 - CBCS 4-56

- CS 4-55
- EDCAEN 4-55
- EDCAST5-0 (EDCA #5-0 status) 4-42
- EDCD (data event detection channel) 4-24, 4-58
 - control register (EDCD_CTRL) 4-58
 - mask register (EDCD_MASK) 4-61
 - reference value register (EDCD_REF) 4-61
- EDCD_CTRL (EDCD control register) 4-58
 - ATS 4-60
 - AWS 4-59
 - CCS 4-60
 - EDCDEN 4-60
- EDCD_MASK (EDCD mark register) 4-61
- EDCD_REF (reference value register) 4-61
- EDCDEN (EDCD enable) 4-60
- EDCDST (EDCD status) 4-42
- EDU (event detection unit) 4-54
 - address event detection channel (EDCA) 4-22, 4-54
 - data event detection channel (EDCD) 4-24, 4-58
- EE pins
 - control register (EE_CTRL) 4-45
- EE_CTRL register
 - EE0DEF 4-47
 - EE1DEF 4-47
 - EE2DEF 4-47
 - EE3DEF 4-46
 - EE4DEF 4-46
 - EE5DEF 4-46
 - EEDDEF 4-46
- EE0DEF (EE0 definition bits) 4-47
- EE1DEF (EE1 definition) 4-47
- EE2DEF (EE2 definition) 4-47
- EE3DEF (EE3 definition) 4-46
- EE4DEF (EE4 definition) 4-46
- EE5DEF (EE5 definition) 4-46
- EEDDEF (EED definition) 4-46
- EI A-163
- EMCR (EOnCE monitor and control register)
 - DEBUGERST 4-42
 - DIS 4-42
 - EDCAST5-0 4-42
 - EDCDST 4-42
 - IME 4-42
 - RCVINT 4-41
 - SWDIS 4-42
 - TBFDM 4-41
 - TRSINT 4-41
- EMR (exception and mode register) 3-7
 - BEM 3-8
 - clearing EMR bits 3-10
 - DOVF 3-8
 - GP6-0 3-8
 - ILIN 3-9
 - ILST 3-9
 - NMID 3-8
- Emulation and debug 4-1
- Endian support 2-56
 - bit mask instructions 2-67
 - change-of-flow instructions 2-68
 - control instructions 2-68
 - data moves 2-58
 - data transfer 2-59
 - instruction word transfers 2-62
 - memory access behavior 2-64
 - multi-register transfer 2-61
 - stack support instructions 2-67
- EOnCE 1-3
- EOnCE (enhanced on-chip emulator) 1-3, 2-5, 4-1, 4-10
 - command registers (ECR) 4-36
 - dedicated instructions 4-11
 - EE pins 4-18
 - internal architecture 4-16
 - register addressing 4-30
 - register addressing offsets 4-31
- EOnCE controller
 - functionality 4-15
 - register set 4-17
- EOnCE controller registers
 - command register (ECR) 4-36
 - core command register (CORE_CMD) 4-48
 - monitor and control register (EMCR) 4-41
 - PC breakpoint detection register (PC_DETECT) 4-49
 - PC of last execution set (PC_LAST) 4-49
 - PC of the exception execution set (PC_EXCP) 4-49
 - PC of the next execution set (PC_NEXT) 4-49
 - receive register (ERCV) 4-43
 - status register (ESR) 4-37
 - transmit register (ETRSMT) 4-43
- EOnCE pins 4-10
- EOR A-165, A-167
- EOR.W A-169
- ES (event selector) 4-25, 4-61
 - control register (ESEL_CTRL) 4-61
 - mask debug exception register (ESEL_DI) 4-64
 - mask debug mode register (ESEL_DM) 4-63
 - mask disable trace register (ESEL_DTB) 4-65
 - mask enable trace register (ESEL_ETB) 4-64
- ESEL_CTRL (ES control register) 4-26
 - SELDI 4-62
 - SELDM 4-62
 - SELDTB 4-62
 - SELETB 4-62
- ESEL_DI (ES mask debug exception register) 4-26, 4-64
- ESEL_DM (ES mask debug mode register) 4-26, 4-63

ESEL_DTB (ES mask disable trace register) 4-26, 4-65
 ESEL_ETB (ES mask enable trace register) 4-26, 4-64
 ESP (exception stack pointer register) 2-35
 ESR (EOnCE status register) 4-37
 CORES 4-38
 CORETP 4-39
 DRCOUNTER 4-40
 DREE4-0 4-39
 DRSW 4-39
 DRTBFULL 4-39
 NOCHOF 4-39
 PCKILL 4-38
 RCV 4-38
 REVNO 4-39
 TBFULL 4-39
 TRSMT 4-38
 ESR register 4-38
 DREDCA7-0 4-40
 Event counter
 control register (ECNT_CTRL) 4-50
 programming model 4-18, 4-50
 value register (ECNT_VAL) 4-52
 Event counter control 4-18
 Event selector
 ESEL_CTRL 4-26
 ESEL_DI 4-26
 ESEL_DM 4-26
 ESEL_DTB 4-26
 ESEL_ETB 4-26
 EX (exit command) 4-37
 Exception
 interface to the pipeline 5-49
 internal exceptions 5-50
 pipeline 5-3, 5-53, 5-56
 Exception processing 5-46
 Execution stage 5-5
 EXP (exception mode bit) 3-4
 EXT 2-9
 EXT (extended mode of operation) 4-51
 EXTRACT A-171
 EXTRACTU A-173

G

GO (go command) 4-37
 GP6-0 (general purpose flags) 3-8
 Grouping 5-5
 assembly reordering rules 5-12
 conditional execution 5-9
 general 5-5
 mechanism 5-6
 prefix words 5-7
 types of
 prefix 5-7
 serial (non-prefix) 5-7

Grouping Mechanism 5-6

I

I2-0 (interrupt mask bits) 3-3
 IADDNC.W A-175
 IFc A-176
 ILIN (illegal instruction) 3-9
 ILLEGAL A-178
 ILST (illegal execution set) 3-9
 IMAC A-180
 IMACLHUU A-183
 IMACUS A-185
 IME (interrupt mode enable) 4-42
 IMPY A-187
 IMPY.W A-189
 IMPYHLUU A-191
 IMPYSU A-193
 IMPYUU A-195
 INC A-197
 INC.F A-199
 INCA A-201
 INSERT A-203
 Instruction bus 2-5
 Instruction dispatch 5-4
 Instruction Grouping
 see Grouping 5-5
 Instruction Set A-19
 ABS A-20
 ADC A-22
 ADD A-24
 ADD2 A-27
 ADDA A-29
 ADDL1A A-32
 ADDL2A A-34
 ADDNC.W A-36
 ADR A-38
 AND A-40, A-43
 AND.W A-45
 ASL A-48
 ASL2A A-50
 ASLA A-51
 ASLL A-52
 ASLW A-55
 ASR A-57
 ASRA A-59
 ASRR A-60
 ASRW A-63
 BF A-65
 BFD A-67
 BMCHG A-69
 BMCHG.W A-69
 BMCLR A-75
 BMCLR.W A-78
 BMSET A-80



BMSET.W A-82
BMTSET A-84
BMTSET.W A-86
BMTSTC A-89
BMTSTC.W A-91
BMTSTS A-94
BMTSTS.W A-96
BRA A-99
BRAD A-101
BREAK A-103
BSR A-105
BSRD A-107
BT A-109
BTD A-111
CLB A-113
CLR A-115
CMPEQ A-117
CMPEQ.W A-119
CMPEQA A-121
CMPGT A-123
CMPGT.W A-125
CMPGTA A-127
CMPHI A-129
CMPHIA A-131
CONT A-133
CONTD A-135
DEBUG A-137
DEBUGEV A-138
DECA A-139
DECEQ A-141
DECEQA A-143
DECGE A-144
DECGEA A-146
DI A-148
DIV A-150
DMACSS A-153
DMACSU A-155
DOEN_n A-157
DOENSH_n A-159
DOSETUP_n A-161
EI A-163
EOR A-165, A-167
EOR.W A-169
EXTRACT A-171
EXTRACTU A-173
IADDNC.W A-175
IF_c A-176
ILLEGAL A-178
IMAC A-180
IMACLHUU A-183
IMACUS A-185
IMPY A-187
IMPY.W A-189
IMPYHLUU A-191
IMPYSU A-193
IMPYUU A-195
INC A-197
INC.F A-199
INCA A-201
INSERT A-203
JF A-205
JFD A-207
JMP A-209
JMPD A-211
JSR A-213
JSRD A-215
JT A-217
JTD A-219
LPMARK_x A-221
LSLL A-224
LSR A-226
LSRA A-227
LSRR A-228
LSRW A-231
MAC A-233
MACR A-236
MACSU A-239
MACUS A-241
MACUU A-243
MARK A-245
MAX A-246
MAX2 A-247
MAX2VIT A-249
MAXM A-251
MIN A-253
MOVE.2F A-254
MOVE.2L A-256
MOVE.2W A-258
MOVE.4W A-262
MOVE.B A-264
MOVE.F A-268
MOVE.L A-272, A-275, A-279
MOVE.W A-285, A-289
MOVE_c A-295
MOVES.2F A-297
MOVES.F A-299
MOVES.L A-301
MOVEU.B A-307
MOVEU.L A-311
MOVEU.W A-313, A-315
MPY A-319
MPYR A-322
MPYSU A-325
MPYUS A-327
MPYUU A-329
NEG A-331
NOP A-333
NOT A-334, A-336

- NOT.W A-338
 - OR A-340, A-342
 - OR.W A-344
 - POP A-347
 - POP.N A-350
 - PUSH A-353
 - PUSHN A-356
 - RND A-359
 - ROL A-362
 - ROR A-364
 - RTE A-366
 - RTED A-368
 - RTS A-370
 - RTSD A-372
 - RTSTK A-374
 - RTSTKD A-376
 - SAT.F A-378
 - SAT.L A-380
 - SBC A-382
 - SBR A-384
 - SKIPLS A-386
 - STOP A-388
 - SUB A-389
 - SUB2 A-392
 - SUBA A-394
 - SUBL A-396
 - SUBNC.W A-398
 - SXT.x A-400
 - SXTA.x A-402
 - TFR A-404
 - TFRA A-406, A-408
 - TFRc A-410
 - TRAP A-412
 - TSTEQ A-414
 - TSTEQA.x A-415
 - TSTGE A-417
 - TSTGEAL A-418
 - TSTGT A-420
 - TSTGTA A-421
 - VSL A-422
 - WAIT A-426
 - ZXT.x A-428
 - ZXTA.x A-430
 - Instruction set accelerator 2-5
 - Instruction Set Accelerator Plug-In 6-57
 - IPL (interrupt priority level) 3-3
 - ISAP 2-5, 6-57
 - allocating encoding space 6-60
 - Conditional execution 6-66
 - How the core identifies ISAP instructions 6-63
 - Programming rules 6-67
 - working with a single ISAP 6-58
 - working with data and memory 6-60
 - Working with multiple ISAPs 6-59
-
- J**
 - JF A-205
 - JFD A-207
 - JMP A-209
 - JMPD A-211
 - JSR A-213
 - JSRD A-215
 - JT A-217
 - JTAG 5-44, 5-45
 - JTAG access 4-33
 - JTAG and EOnCE interface 4-2
 - JTAG interface pins 4-2
 - JTD A-219
-
- L**
 - LF3-0 (loop flags 3-0) 3-2
 - Linear addressing mode 2-45
 - Loop
 - looping rules 5-32
 - nested loop 5-31
 - timing 5-32
 - LPMARKx 5-26, A-221
 - LSLL A-224
 - LSR A-226
 - LSRA A-227
 - LSRR A-228
 - LSRW A-231
-
- M**
 - M0-M3 (modifier registers) 2-36
 - MAC A-233
 - MAC (multiply-accumulate) 1-3, 2-3, 2-10
 - MAC unit
 - arithmetic instructions 2-10
 - MACR A-236
 - MACSU A-239
 - MACUS A-241
 - MACUU A-243
 - MARK A-245
 - MAX A-246
 - MAX2 A-247
 - MAX2VIT A-249
 - MAXM A-251
 - MCTL (modifier control register) 2-37
 - MCTL register
 - AM bits 2-37
 - Memory
 - on-chip 2-5
 - Memory access
 - behavior in big/little endian modes 2-64
 - memory access misalignment 2-42
 - Memory interface 2-55
 - Memory organization 2-57

MIN A-253
 Modifier registers (M0-M3) 2-36
 Modulo adder 2-33
 Modulo addressing 2-4
 Modulo addressing mode 2-45
 Move instructions 2-51, 2-52
 fractional moves 2-54
 integer moves 2-53
 MOVE.2F A-254
 MOVE.2L A-256
 MOVE.2W A-258
 MOVE.4W A-262
 MOVE.B A-264
 MOVE.F A-268
 MOVE.L A-272, A-275, A-279
 MOVE.W A-285, A-289
 MOVEc A-295
 MOVES.2F A-297
 MOVES.F A-299
 MOVES.L A-301
 MOVEU.B A-307
 MOVEU.L A-311
 MOVEU.W A-313, A-315
 MPY A-319
 MPYR A-322
 MPYSU A-325
 MPYUS A-327
 MPYUU A-329
 Multiple wrap-around modulo addressing mode 2-47
 Multiplication 2-20
 Multiply-accumulate (MAC) 1-3
 Multiply-accumulate (MAC) unit 2-10
 Multi-precision arithmetic support 2-28, 2-30

N

N0-N3 (offset registers) 2-36
 NEG A-331
 Nested loop 5-31
 NMI (non-maskable interrupts) 5-50
 NMID (NMI disable bit) 3-8
 NOCHOF (no CHOF in debug mode) 4-39
 NOP A-333
 definition 7-60
 NOT A-334, A-336
 NOT.W A-338
 NSP (normal stack pointer register) 2-35

O

Offset adder 2-4
 Offset registers (N0-N3) 2-36
 OR A-340, A-342
 OR.W A-344
 OVE (overflow exception enable bit) 3-4

P

PAB (program address bus) 2-1
 PAG (program address generator) 2-5
 PC (program counter) 1-3, 2-5
 PC relative addressing modes 2-40
 PC relative mode 2-40
 PC_DETECT (PC breakpoint detection register) 4-49
 PC_EXCP (PC of the exception execution set) 4-49
 PC_LAST (PC of last execution set) 4-49
 PC_NEXT (PC of the next execution set) 4-49
 PCKILL (PC killed) 4-38
 PCU (program control unit) 2-5
 PDB (program data bus) 2-1
 PDU (program dispatch unit) 2-5
 Pipeline 5-1
 address generation 5-4
 execution 5-5
 instruction dispatch 5-4
 instruction pre-fetch and fetch 5-4
 stages 5-2, 5-3
 POP A-347
 POPN A-350
 Power saving considerations 4-16
 Pre-fetch and fetch stages 5-4
 Prefix Grouping 5-7
 Prefix word encoding A-7
 Processing states 5-41
 debug mode 5-44
 exception 5-37
 reset 5-43
 stop 5-45
 wait 5-44
 Program control 5-1
 Program control instructions 5-41
 Programming rules
 ISAP-specific 6-67
 PSEQ (program sequencer and control unit) 1-3, 2-5
 PSEQ (program sequencer unit) 2-5
 PUSH A-353
 PUSHN A-356

R

R/W (read or write command bit) 4-36
 R0-R7 registers 2-35, 2-37
 RCV (receive) 4-38
 RCVINT (receive interrupt) 4-41
 Register direct addressing modes 2-38
 Register indirect addressing modes 2-38
 REGSEL (register select) 4-37
 Reset processing state 5-43
 Reverse-carry addressing mode 2-45
 REVNO (revision number) B-1
 bits 4-39

- RM (rounding mode bit) 3-5
- RND A-359
- ROL A-362
- ROR A-364
- Rounding 2-21, 2-23
- RTE A-366
- RTED A-368
- RTS A-370
- RTSD A-372
- RTSTK A-374
- RTSTKD A-376
- S**
- S (scaling bit) 3-5
- S1-0 (scaling mode bits) 3-5
- SAT.F A-378
- SAT.L A-380
- SBC A-382
- SBR A-384
- Scaling bit 3-5
- Scaling mode bits 3-5
- SCID B-1
 - bit assignments B-2
 - fields B-1
- SCR (StarCore registry) B-1
- SELDI (selection bit for debug exception) 4-62
- SELDM (selection bit for debug mode) 4-62
- SELDTB (selection bit for disable trace) 4-62
- SELETB (selection bit for enable trace) 4-62
- Semaphore support instructions 2-50
- Semaphoring 2-4
- Serial Grouping 5-7
- Shadow stack pointer registers 2-36
- Sign extension 2-8
- Signed fractional data format 2-18
- Signed integer data format 2-19
- SKIPLS A-386
- SLF (short loop flag) 3-2
- SM (arithmetic saturation mode) 3-6
- Software downloading 4-12
- Software stack 5-34
- SP (stack pointer) registers 2-35
- Special address modes 2-41
- SR (status register) 3-1, 3-2
 - C 3-7
 - DI 3-4
 - EXP 3-4
 - I2-0 3-3
 - LF3-0 3-2
 - OVE 3-4
 - RM 3-5
 - S 3-5
 - S1-0 3-5
 - SLF 3-2
 - SM 3-6
 - T 3-6
 - VF3-0 3-4
- SRAM (static random access memory) 1-2
- Stack pointer registers 2-4
- Stack support 5-32
 - fast call-return from subroutines 5-36
 - instructions 2-67
 - normal and exception modes 5-32
 - shadow stack pointer registers 5-35
- STOP A-388
- Stop processing state 5-45
- SUB A-389
- SUB2 A-392
- SUBA A-394
- SUBL A-396
- SUBNC.W A-398
- SWDIS (software access disable) 4-42
- SXT.x A-400
- SXTA.x A-402
- T**
- T (true bit) 3-6
- TB_BUFF (trace buffer) 4-29, 4-69
- TB_CTRL (trace buffer control register)
 - TCHOF 4-68
 - TCNTEXT 4-67
 - TCOUNT 4-67
 - TEN 4-68
 - TEXEC 4-68
 - TINT 4-68
 - TLOOP 4-68
 - TMARK 4-68
- TB_RD (read pointer register) 4-29, 4-69
- TB_WR (write pointer register) 4-69
- TBFDL (enter debug on trace buffer full) 4-41
- TBFULL (trace buffer full) 4-29, 4-39
- TCHOF (trace addresses of change of flow instructions enable mode) 4-68
- TCK (test clock input pin) 4-2
- TCNTEXT (trace buffer extension counter mode) 4-67
- TCOUNT (trace buffer counter mode) 4-67
- TDI (test data input pin) 4-2
- TDO (test data output pin) 4-2
- TEN (trace buffer counter mode) 4-68
- TEXEC (trace issue of execution sets enable mode) 4-68
- TFR A-404
- TFRA A-406, A-408
- TFRC A-410
- TINT (trace interrupts enable mode) 4-68
- TLOOP (trace loops mode) 4-68
- TMARK (trace mark instruction mode) 4-68
- TMS (test mode select input pin) 4-2



Trace unit

- control register (TB_CTRL) 4-65
- read pointer register (TB_RD) 4-69
- register set 4-30
- virtual register (TB_BUFF) 4-69
- write pointer register (TB_WR) 4-69

TRAP 5-37, A-412

TRSINT (transmit interrupt) 4-41

TRSMT (transmit) 4-38

TRST (test reset pin) 4-2

True bit 3-6

TSTEQ A-414

TSTEQA.x A-415

TSTGE A-417

TSTGEA.L A-418

TSTGT A-420

TSTGTA A-421

Two's complement rounding 2-23

U

Unsigned arithmetic 2-20

Unsigned integer data format 2-19

Unsigned multiplication 2-20

V

VF3-0 (Viterbi flags 3-0) 3-4

Viterbi decoding support 2-30

VLES (variable length execution set) 1-2

VSL A-422

W

WAIT A-426

Wait processing state 5-44

X

XABA and XABB (data memory address buses) 2-1

XDBA and XDBB (data memory address buses) 2-1,
2-6

Z

ZXT.x A-428

ZXTA.x A-430





