DSP56800 to DSP56800E

Porting Guide

Digital Signal Controllers

DSP56800ERG
Rev. 1.1
11/2005

freescale.com

Contents

List of Figures

List of Tables

List of Code Examples

Chapter 1
Introduction

The DSP56800E is designed to enhance the combined MCU/DSC functionality of its predecessor, the DSP56800. The new architecture has been developed to ensure that programs originally written for the DSP56800 (also referred to as legacy code) will assemble and run correctly through the porting process. This process can be guaranteed by adopting a number of simple rules described in this guide. The CodeWarrior Assembler (CW) available from Metrowerks, also provides messages designed to facilitate the porting process.

The complete instruction set of the DSP56800 is supported by the toolset available for the DSP56800E. Assembly code developed for the DSP56800 can be directly interpreted by the CW Assembler when the option to allow legacy instructions is set. When this mode of operation is enabled, the assembler is configured to recognize the DSP56800 as well as DSP56800E syntax. Consequently, the original source code generally does not have to be translated to the new DSP56800E syntax unless the user wants to optimize the code for performance and code density. Debugging assembly files under CodeWarrior is also done on the original syntax.

The new architecture also introduces additional addressing modes and instructions for the expressed purpose of supporting legacy DSP56800 instructions `LEA`, `TSTW`, and `MOVE`. This added capability is not described in the *DSP56800E Core Reference Manual*; it is only documented in this guide. These special legacy instructions are only preserved in the 56800E's architecture to guarantee exact execution of legacy code. The legacy instructions appear shaded in Table A-10 page A-8 to specifically highlight their use in this architecture.

Similarly, there are some differences in instruction syntax between the DSP56800 and DSP56800E architectures which are shown throughout the guide. The exact mnemonic mapping of each DSP56800 instruction into the corresponding DSP56800E assembly instruction is also shown in Table A-10 through Table A-14. In some cases, the mapping requires an adjustment, as is the case when loading the N register, (see Note 1 at the end of Table A-10), or when replacing `REP LC` with the `DOSLC` instruction.

This guide discusses all porting issues between the DSP56800 and the DSP56800E. It illustrates and examines architectural differences, discusses all compatibility issues between the two architectures, and provides assembly code segments which demonstrate how compatibility can be achieved with minimal user intervention on the original legacy code.

Using This Document

This document provides a guide for porting legacy DSP56800 code to execute in the DSP56800E core. It is organized to provide the following information:

- Quick reference on compatibility issues, page 2-1
- Summary of the architectural extensions in the DSP56800E, page 3-1
- Programming model comparison, page 3-2
- Memory model comparison, page 3-4
- AGU registers, page 4-1
- Compatibility issues, page 5-1
- Tuning ported code for performance, page 6-1
- Mapping tables from DSP56800 to DSP56800E architecture, page A-8

Running DSP56800 Code on the DSP56800E

To use this document efficiently, the user should read Chapter 2, “Quick Reference”. Chapter 2 presents a summary of all compatibility issues that may arise during the porting process, and provides a list of recommendations for producing upward-compatible code for the DSP56800.

Understanding Architectural Differences

To understand the architectural differences between the DSP56800 and the DSP56800E, users should familiarize themselves with Chapter 3, “Comparing the Two Architectures”. This chapter introduces the extended features of the DSP56800E that enhance performance and improve program code density. Further information on DSP56800E architecture can be found in the *DSP56800E Core Reference Manual*.

Tuning Ported Code

Once the code has been assembled correctly, the user can refer to Chapter 6, “Optimizing Legacy Code” for hints on optimizing performance and reducing code density. Chapter 6 does not provide a complete discussion of this issue because it is dependent on the application and programming style.

Chapter 2
Quick Reference

This chapter summarizes the compatibility issues that can arise between the DSP56800 and DSP56800E and offers a list of recommendations for producing DSP56800 that is compatible with the DSP56800E.

2.1 Compatibility Issues

This section provides a quick reference list of possible compatibility issues between the DSP56800 and DSP56800E.

1. Section 2.1.1 lists all cases that require immediate user intervention.
2. Section 2.1.2 on page 2-4 presents uncommon cases where intervention is not immediately required.
3. Section 2.1.3 on page 2-6 covers cases where intervention is not needed.
4. Section 2.1.4 on page 2-7 describes cases that may arise when both syntax are used together.

All items are cross-referenced with their detailed discussion in Chapter 5, "Compatibility Issues."

2.1.1 Intervention Required

The compatibility issues summarized in Table 2-1 include all cases which may require some intervention from the user. In some cases the mapping of the assembler will generate the desired behavior. But in cases where the programming style is incompatible with the new architecture, the user may be required to make some minor adjustments to the legacy code, e.g. replacing numerical targets with labels, or sign extending the N register when its actually used as an offset (rather than a pointer).

Table 2-1. Cases With User-Intervention Requirements

Case	Descriptions
UI-1	`Jcc` <Numerical_Absolute_Address> `JSR` <Numerical_Absolute_Address> `JMP` <Numerical_Absolute_Address> `DO` <Source>,<Numerical_Absolute_Address>
	Assembler requires labels for Absolute target addresses. Guarantees correct operation in cases where code growth occurs and intended target moves. Refer to Section 5.9 on page 5–10.

Table 2-1. Cases With User-Intervention Requirements (Continued)

Case	Descriptions
UI-2	`Bcc` *(±)<Numerical_Relative_Offset> (or without the “*”) `BRA` *(±)<Numerical_Relative_Offset> (or without the “*”) `BRCLR` #Mask8,<Source>,*(±)<Numerical_Relative_Offset> (or w/o “*”) `BRSET` #Mask8,<Source>,*(±)<Numerical_Relative_Offset> (or w/o “*”)
	Assembler requires labels for target addresses. Guarantees correct operation in cases where code growth occurs and intended target moves. Refer to Section 5.9 on page 5–10.
UI-3	`Bcc` <Target_Label> `BRA` <Target_Label>
	If code growth places the Target_Label beyond the 7-bit offset, a link error will occur; the user must use the forcing operator “>” on the Target_Label to force the assembler to use a branch with an 18-bit offset. Refer to Section 5.4.3 on page 5–6.
UI-4	`BRCLR` #Mask8,<Source>,<Target_Label> `BRSET` #Mask8,<Source>,<Target_Label>
	If code growth places the Target_Label beyond the 7-bit offset, a link error will occur. Only 7-bit offset are available for these instructions. Replace `BRCLR` with “`BFTSTL #MASK16`,<Source>” followed by “`BCS` ><Target_Label”. For `BRSET`, use the instruction “`BFTSTH`” followed by `BCS`. Refer to Section 5.4.3 on page 5–6.
UI-5	`DO` `LA`, <End_of_Loop_Label>
	The LA register is no longer supported for this instruction; use an alternate register. Refer to Section 5.10.1 on page 5–11.
UI-6	`REP LC`
	User must map instruction to `DOSLC` and insert a single `NOP` instruction in the loop body to complete the 2-word minimum requirement for `DOSLC` instruction. Refer to Section 5.10.1 on page 5–11
UI-7	`REP LA`
	The LA register is no longer supported for this instruction; use an alternate register. Refer to Section 5.10.1 on page 5–11.

UI-8	`LEA (SP)+N`
	`MOVE X:(SP+N),8-HHHHH`
	`MOVE X:(SP+N),8-SSSS`
	`MOVE 8-DDDDD,X:(SP+N)`
	`TSTW X:(SP+N)`

	For register indirect addressing modes using both the SP and N registers, the instruction “`SXTA.W N`” must immediately precede any of the instructions listed above, (to preserve the intended sign). Note: If N is not used as an offset register, no sign extension is required, (the upper 8-bits should be zeroes). Refer to Section 5.4.2 on page 5–5.

Table 2-1. Cases With User-Intervention Requirements (Continued)

Case	Descriptions
UI-9	`MOVE <Source_Register>,X:(Rj+xxxx)` (R0 & R1 in modulo addressing only) `MOVE X:(Rj+xxxx),<Dest_Register>` (R0 & R1 in modulo addressing only) `TSTW X:(Rj+xxxx)` (R0 & R1 in modulo addressing only)

	If modulo addressing is active and `Rj = {R0,R1}` is the base address where xxxx is negative and represents the offset, above instructions must be mapped to 56800E equivalent, `MOVE.W` and `TST.W`. When the original xxxx is represented in hex, it must be sign extended to 32-bits, (e.g. $FFF8 should be replaced by $FFFF FFF8). If instead, `xxxx` represents the base address and `Rj` the offset, the following sequence must be utilized to remain compatible:
	Examples:
	`LEA (Rj+xxxx)` `LEA (Rj+xxxx)`
	`MOVE X:(Rj),<Dest_Register>` `TSTW X:(Rj)`
	`LEA (Rj-xxxx)` `LEA (Rj-xxxx)`
	Refer to Section 5.3 on page 5–3.

UI-10	New requirements when entering and exiting X/P Mode.
	Specific sequence must be followed to switch to (and return from) executing programs from data memory. Refer to Section 5.14 on page 5–14.
UI-11	Compatibility issues at the chip level
	• Priority levels have increased from two levels to four levels in the DSP56800E. • The location of the interrupt vector is not restricted by the core. For exact compatibility, user must adjust references to the table. • The location of the peripheral space is not restricted by the core. For exact compatibility, the location of the memory mapped register should be in the exact locations as the DSP56800 chip implementations. • The DSP56800E architecture does not restrict the performance of dual read operations on off-chip data memory. • The DSP56800E core does not define the exact operation of the OMR's EX bit. (When this bit is set on the DSP56800, it forces all accesses to X memory to be from off-chip X data memory). Refer to Section 5.18 on page 5–21.
UI-12	`BFCLR #$0300,SR` ; Enable Interrupts `BFSET #$0300,SR` ; Disable Interrupts
	There is a delay which occurs between the execution of the BFCLR instruction and the point where the interrupt arbiter sees the current core interrupt priority level (CCPL). When interrupts are enabled, instructions in the following 6 clock cycles will be executed before any pending interrupts recognize the new CCPL and are taken. Refer to Section 5.19.1 on page 5–23. There is a delay which occurs between the execution of the BFSET instruction and the point where the interrupt arbiter masks incoming interrupts. When interrupts are disabled, interrupts can still be taken after any instruction completing execution during any of the next 5 cycles following the BFSET instruction. Instruction beginning execution in the 6th clock cycle, form the beginning of a non-interruptible sequence. Refer to Section 5.19.2 on page 5–24.

2.1.2 Uncommon Cases: Intervention May Be Required

Table 2-2 lists compatibility issues that are uncommon in application code but may require some intervention.

Table 2-2. Uncommon Cases - May Require User-Intervention

Case	Descriptions

UC-1	`REP #xx`
	`REP <Register>`

	A number of instructions that are not commonly used in repeat loops can no longer be repeated due to code growth. Instead, a `DO` loop must be used to replace the `REP` instruction. Refer to Section 5.10.3 on page 5–12.
UC-2	`MOVE <Source>,LC`
	The LC register is 13-bits on the DSP56800, but is 16-bits on the DSP56800E. When reading the 56800's 13-bit LC register, the upper 3-bits are always zeroed. If this feature is important in an application, refer to Section 5.8 on page 5–10.

UC-3	`ADC Y,F`
	`SBC Y,F`
	`DIV DD,F`
	`IMPY <Src1>,<Src2>,FDD (or IMPY16)`

	These instructions are affected and behave differently when SA = 1. On the DSP56800E, these instructions do not saturate their results even if SA is set. Refer to Section 5.11 on page 5–12.
UC-4	`DEC F` (or `DECW`) `INC F` (or `INCW`)
	These instructions are affected and behave differently when CC (or CM in DSP56800E) = 1when the destination is an accumulator. In the 56800E, the zero condition is calculated using bits [31:16] of the accumulator, whereas the 56800 calculates the zero condition using [31:0]. Refer to Section 5.12 on page 5–14.

UC-5	`ADD <Source_Register>,X:aa`
	`ADD <Source_Register>,X:xxxx`
	`ADD <Source_Register>,X:(SP-xx)`

	These instructions are affected when memory is the destination. In the DSP56800E, the carry condition is calculated from bit 31 of the result, whereas in the DSP56800, it is calculated from bit 35 of the result. Refer to Section 5.13 on page 5–14.

Table 2-2. Uncommon Cases - May Require User-Intervention (Continued)

Case	Descriptions

UC-6	Addressing mode, (`R2+xx`) removed, (where `xx` is a 6-bit positive offset):
	`BFCHG   #xxxx,X:(R2+xx)`
	`BFCLR   #xxxx,X:(R2+xx)`
	`BFSET   #xxxx,X:(R2+xx)`
	`BFTSTH  #xxxx,X:(R2+xx)`
	`BFTSTL  #xxxx,X:(R2+xx)`
	`BRSET   #xxxx,X:(R2+xx),`<Target_Label>
	`BRCLR   #xxxx,X:(R2+xx),`<Target_Label>
	`MOVE    #xxxx,X:(R2+xx)`

	Assembler will map these instructions with the Standard 56800E (`Rn+xxxx`) addressing mode. An additional opcode word is required for these instructions. Address wrapping is not performed. Refer to Section 5.2 on page 5–2.
UC-7	Differences in AGU overflow and underflow, e.g. 64K address wrapping.
	Address wrapping beyond 64K word boundary is not recommended and it is considered a flawed non-portable programming style. There are legacy 56800E instructions that guarantees exact behavior when {R0,R1,R2,R3} are used in linear addressing mode, but the following modes can be corrected: `(Rn)+`, `(Rn)-`, `(Rn)+N`, `(R2+xx)`, but `(SP-xx)` is not supported for address wrapping. Refer to Section 4.2 on page 4–23.

UC-8	Bit Manipulation operations on SR bits [14:10]
	`BFCHG   #7Cxx,SR`
	`BFCLR   #7Cxx,SR`
	`BFSET   #7Cxx,SR`
	`BFTSTH  #7Cxx,SR`
	`BFTSTL  #7Cxx,SR`
	`BRSET   #7Cxx,SR,`<Target_Label>
	`BRCLR   #7Cxx,SR,`<Target_Label>

	Bitfield operations on bits [14:10] are not permitted on the DSP56800E architecture. These bits represent the upper 5-bits of the PC register. These same bits are reserved on the DSP56800.
UC-9	Legacy programs growing beyond the 64K word boundary.
	The use of both DSP56800 and DSP56800E instructions is only supported for applications fitting in 64K word program memory space and 64K word data memory space. Refer to Section 5.17 on page 5–20.

UC-10	`LEA     (Rj)+,Ri`
	`LEA     (Rj)-,Ri`
	`LEA     (Rj)+N,Ri`
	`LEA     (Rj+xxxx),Ri`
	For `Rj = Ri = {R0,R1,R2,R3}`

	Unsupported DSP56800 `LEA` Instruction Syntax. This undocumented syntax may be accepted by the toolset but is not supported by the DSP56800E Specifications. Refer to Section 5.15 on page 5–17.

2.1.3 No User Intervention Required

Table 2-3 lists compatibility issues that are handled automatically by the toolset and do not require intervention.

Table 2-3. Other Cases—No User-Intervention Required

Case	Descriptions
NI-1	`DO LC,<End_of_Loop_Label>`
	If the body of the original loop is 1-word of length, the assembler inserts a single `NOP` instruction in the loop body to complete the 2-word minimum requirement for the `DOSLC` instruction. The assembler also remaps the instruction to `DOSLC`. Refer to Section 5.10.1 on page 5–11.

NI-2	Addressing mode, `(R2+xx)` removed, (where `xx` is a 6-bit positive offset):
	`MOVE X:(R2+xx),8-HHHH`
	`MOVE 8-HHHH,X:(R2+xx)`
	`TSTW X:(R2+xx)`
	`LEA (R2+xx)`

	Assembler will map these instructions with the Legacy 56800E (`Rn+xxxx`) addressing mode. An additional opcode word is required for these instructions. Address wrapping is performed. Refer to Section 5.2 on page 5–2.

NI-3	`ADD X:aa,FDD`
	`SUB X:aa,FDD`
	`CMP X:aa,FDD`
	`DECW X:aa`
	`INCW X:aa`
	`MOVE X:(SP-xx),F1`
	`MOVE F,X:(SP-xx)`
	`MOVE #xx,Rj` (with immediate value between [-64,-1])
	`MOVE #xx,F1`
	All Instructions using the (`R2+xx`) addressing mode.

	These DSP56800 instructions are automatically mapped to the 56800E equivalent with one extra instruction word. Refer to Section 5.4.5 on page 5–7.

NI-4	`LSLL Y1,X0,DD`
	`LSLL Y0,X0,DD`
	`LSLL Y1,Y0,DD`
	`LSLL Y0,Y0,DD`
	`LSLL A1,Y0,DD`
	`LSLL B1,Y1,DD`
	`ASL DD`
	`CLR {X0, Y1, Y0, A1, B1, R0-R3, or N}`
	`POP 8-DDDDD`
	`POP`
	All Instructions above are aliases that were originally mapped to 56800 instructions.

	In all the above cases, the DSP56800E assembler will correctly recognize and map the instruction to its equivalent. Refer to Section A.4, “DSP56800 Instruction Aliases,” on page A-25.

2.1.4 Mixed 56800 & 56800E Instructions

Table 2-4 lists compatibility issues that result from mixing syntax from both architectures in an attempt to optimize for performance and code density.

Table 2-4. Mixed DSP56800 & DSP56800E Syntax

Case	Descriptions
MX-1	Sign Extension Requirements for {R0-R3, N} when mixing with 56800E instructions
	AGU load instructions are mostly mapped to `MOVEU.W`. This ensures that the upper 8 bits are zeroes, which correctly maps the nature of address values in the 56800. However, when the value in the register represents an offset and code is mixed with 56800E instructions, then sign extension may be required. Refer to Section 5.7 on page 5–9.
MX-2	`MOVE` <Source_Register>,`X:(SP)+` MOVE `X:(SP)-`,<Source_Register>
	If legacy code is mixed with new DSP56800E instructions and `MOVE.L` is used to push and pop from the stack (to optimize for performance and codesize), the SP must always remain odd-aligned. In this scenario, if a long move is used and SP is even, a misaligned data access non-maskable interrupt is generated. There is no issue when porting DSP56800 instructions only. Refer to Section 5.16 on page 5–17.
MX-3	New requirements in stack pointer alignment (in Mixed code only).
	Long moves must always point to the even address (where the lower 16-bits are stored), except when the stack is accessed. If legacy code is upgraded to use long moves, proper care must be taken to keep SP odd-aligned. Refer to Section 5.16 on page 5–17.
MX-4	New requirements in context save and restore.
	When storing registers in the DSP56800E, care must be taken when storing LC and HWS. The following rules must be followed: 1. If LC must be stored, store LC2 before LC to guarantee restoration of LC2. 2. If HWS must be stored, store SR and OMR before HWS. 3. When storing HWS, two stores are required to store HWS0 followed by HWS1. 4. When storing 36-bits accumulator with `MOVE.L`, store extension register first. (Rules 2 & 3 are also recommended in the DSP56800). Refer to Section 5.16 on page 5–17.

2.2 Coding Recommendations

The following is a list of recommendations for writing DSP56800 programs that are compatible with the DSP56800E

- Applications should be written so that there is no AGU overflow or underflow over 64K boundaries when accessing data or program memory.
- Do not use program memory space above 64K. For applications that need more than 64K words, it is necessary to use the DSP56800E instruction set only.
- Address wrapping with the stack pointer is not permitted.
- Interpretation of use for AGU registers is important. If an AGU register is used as an offset, it requires a 24-bit signed value. If it represents a pointer, the upper 8-bits must be zeroes.
- Keep the stack pointer odd-aligned when long data elements are stored onto the stack or retrieved from the stack with the MOVE.L instruction.
- Avoid using the DSP56800 instruction aliases (refer to Section A.3 on page A-24).
- Don’t set the CC bit in the OMR register; the DSP56800E architecture defines all data types, and condition codes can be set based on the size of the operand.
- Use labels for all branch and jump instructions rather than numerical targets to protect against problems due to growth in code size.
- Do not use an ENDDO instruction in a repeat loop.
- Do not use the information in condition codes updated by the ENDDO instruction.
- In a repeat loop:
  - — Do not write to the M01 register.
  - — Do not access the LC register.
  - — Do not use `A1,Y0,A` or `B1,Y1,B` as operands for a multiply, shift, or multiply-accumulate instruction.
- When the recognize-legacy-code switch is set, the assembler automatically selects memory switches to recognize only 64K words of data space and 64K words of program space.

Chapter 3
Comparing the Two Architectures

There are several fundamental changes to the architecture which are useful to understand in the porting process:

- The AGU registers and the AGU arithmetic units have grown from 16 to 24 bits.
- The program memory space has expanded to 4MB.
- The data memory space has grown to 32MB.
- Mapping of some DSP56800 instructions to DSP56800E counterparts require one extra instruction word.
- There are some changes to the pipeline.

The following sections summarize the new features of the DSP56800E, including comparisons of the programming model and memory model between the two architectures. For a complete description of each performance enhancing feature, refer to the *DSP56800E Core Reference Manual*.

3.1 Extending the DSP56800 Architecture

The DSP56800E core architecture extends Freescale’s DSP56800 Family architecture to a new generation. It remains source-code compatible with DSP56800 devices and adds the following new features:

- Byte and long data types, supplementing the DSP56800’s word data type
- 24-bit data memory address space
- 21-bit program memory address space
- Three additional 24-bit pointer registers, R4, R5, and N. The N register can function either as an offset register or a pointer register.
- Four shadow registers, including three 24-bit pointer registers and the Modifier register, M01
- A secondary 16-bit offset register to further enhance the dual parallel data ALU instructions
- Two additional 36-bit accumulator registers
- Full-precision integer multiplication
- 32-bit logical and shifting operations
- The second read in a dual read instruction can now access off-chip memory
- Loop count (LC) register extended to 16 bits
- Full support for nested DO looping through additional loop address and loop count registers, LA2 and LC2
- Loop address and hardware stack extended to 24 bits

- Two additional interrupt priority levels with a software interrupt trap for each level, plus a low priority software trap, `SWILP`
- Eight stage instruction pipeline, resulting in higher execution throughput
- Enhanced On-Chip Emulation (Enhanced OnCE) with three debugging modes:
  - — Non-intrusive real-time debugging
  - — Minimally intrusive real-time debugging
  - — Breakpoint and step mode (core is halted)

3.2 Programming Model Comparison

The programming model of the DSP56800E core is a superset of the programming model of the DSP56800. Figure 3-1 on page 3-3 shows the DSP56800 programming model overlaying the DSP56800E programming model. The shaded areas indicate the extensions on the DSP56800E architecture. These extensions are discussed in this section.

There are some differences to note between the two programming models that may affect currently written legacy code for the DSP56800:

- The loop counter register, LC, is expanded from 13 bits to 16 bits.
- The loop address register, LA, is expanded from 16 bits to 24 bits.
- The pointer registers, R0–R3, are expanded from 16 bits to 24 bits.
- The offset register, N, is expanded from 16 bits to 24 bits. N can also function as a pointer register for instructions using indirect addressing modes.
- The LIFO hardware stack registers, HWS0 and HWS1, (designated as HWS) are expanded from 16 bits to 24 bits.
- The program counter, PC, is expanded from 16 bits to 21 bits.
- The upper 5 bits of the program counter reside in the status register, SR.

There are other differences between the two programming models that will not affect legacy code. The following new resources were not present in the DSP56800 programming model:

- In the address generation unit:
  - — The AGU register file has two new pointer registers—R4, and R5 . In addition, the offset register N can also be used as a pointer register in the DSP56800E.
  - — N3 is a new offset register that provides additional addressing modes for dual parallel instructions.
  - — There are 4 new shadow registers—R0, R1, N, and M01. These registers are used during fast interrupt operation. For additional information on fast interrupt operation, refer to Chapter 8, “Program Controller”, in the *DSP56800E Core Reference Manual*.
- In the data arithmetic logic unit:
  - — The DALU register file has two new accumulators: C and D.
- In the program control unit:
  - — A second loop counter, LC2 and a second loop address register, LA2, have been added to support nested hardware looping
  - — The FIRA and FISR registers are new resources specifically designed to support fast interrupt operation.

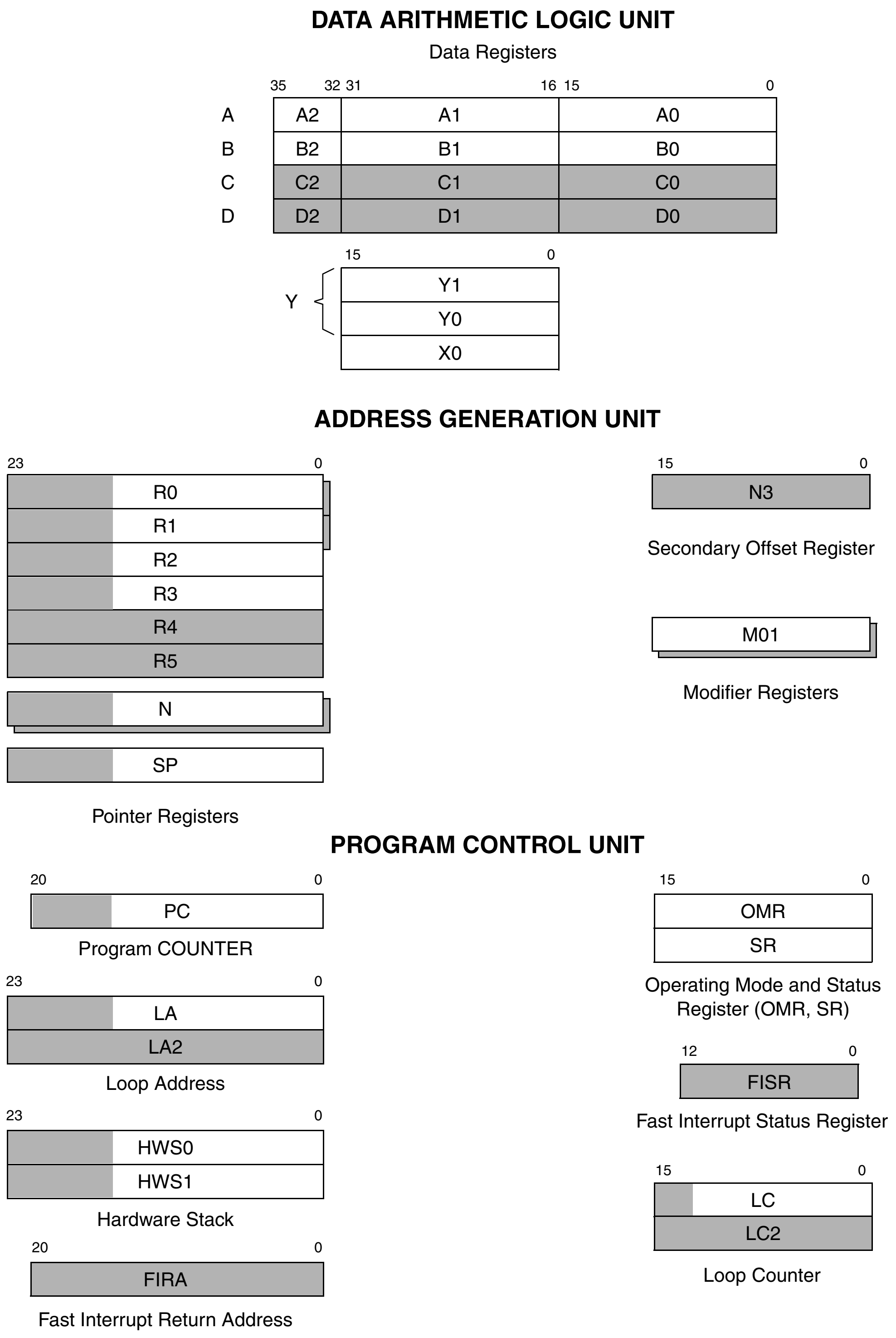

Figure 3-1. Programming Model—DSP56800 vs DSP56800E

3.3 Memory MAP Comparison

The DSP56800 and DSP56800E both contain dual Harvard architectures with separate program and data memory spaces. The amount of addressable program and data memory is different in each device, as shown in sTable 3-1.

Table 3-1. Size of Memory Spaces

Architecture	Program Memory	Data Memory
DSP56800	2^{16}	2^{16}
DSP56800E	2^{21}	2^{24}

3.3.1 DSP56800 Memory Map

The DSP56800 memory map is shown in Figure 3-2.

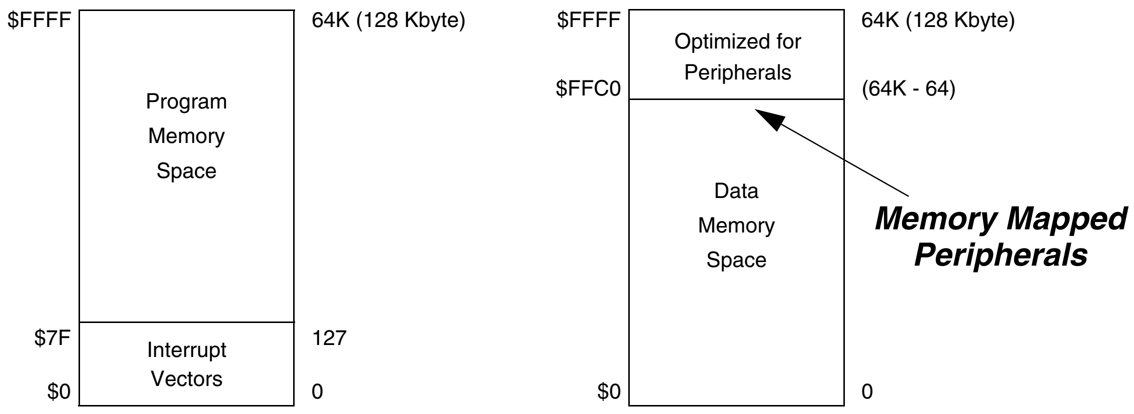

Figure 3-2. DSP56800 Memory Spaces

The DSP56800 program memory map includes the following characteristics:

- Program size is limited to 64K words.
- Data memory space is limited to 64K words.
- The interrupt vector table is fixed (location and size are defined by chip implementation).
- A special peripheral space accessible using the `X:<<pp` addressing mode is fixed at address $FFC0–$FFFF, is limited to 64 words, and is controlled by the chip implementation. (However, peripherals can be located anywhere in the data memory space.)
- The first 64-word block of data memory is accessible using the absolute short addressing mode, `X:aa` (or `X:<aa`).
- No data memory address can be calculated that is larger than 64K.

3.3.2 DSP56800E Memory Map

The DSP56800E memory map is shown in Figure 3-3.

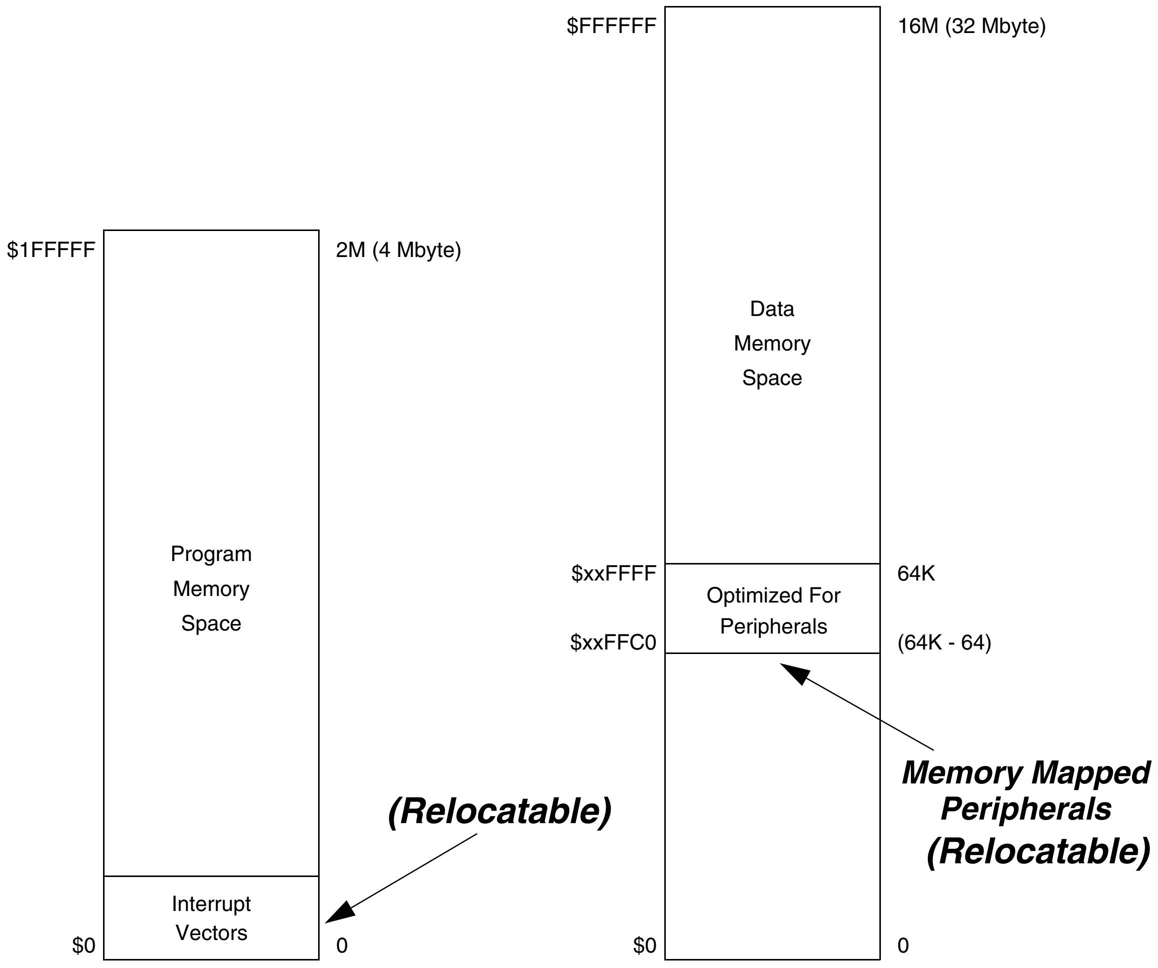

Figure 3-3. DSP56800E Memory Spaces

The DSP56800E program memory map includes the following characteristics:

- Program size is expanded to 2M words (4MB).
- Data space is expanded to 16M words (32MB).
- The interrupt vector table is relocatable (location and size are defined by chip implementation).
- The special peripheral space is relocatable. It is still limited to 64 contiguous words, with location defined by chip implementation.
- The first 64-word block of data memory is accessible using absolute short addressing mode: `X:aa` (or `X:<aa`).

When an application written in DSP56800 program code is translated to the DSP56800E, the new application must fit in the first 64K words of program memory and must only access data memory in the first 64K words of the data memory space. Any growth in program code size must not increase the size of the application above 64K in DSP56800E code. If an application uses both DSP56800 and DSP56800E instructions and no longer fits within 64K, the user must translate the application to the 56800E syntax.

Two exceptions are allowed in the data space. Both the system stack and the peripheral space can be located anywhere within the 24-bit data space range, with peripheral space defined by the chip implementation.

NOTE:

— A compiler can only access the lower 16MB (2^{24}) of Data Memory Space.

— The upper 16MB of data memory cannot be accessed with the following instructions:

- all `MOVE.BP` instructions
- any `MOVE.B` that uses addressing mode `X:(Rn+xxxxxx)` where `xxxxxx` represents a base address in the upper 16MB.

Chapter 4
AGU Registers

To translate a program from the DSP56800 to the DSP56800E, special consideration must be given to the initialization and arithmetic operations of the AGU registers.

4.1 Initializing AGU Registers

There are several rules that must be followed to ensure compatibility of DSP56800 programs when writing to the Program Control and AGU registers.

- Registers R0–R3, HWS, and LA must be zero extended in bits [23:16] and are written using the `MOVEU.W` instruction when writing from another register or from memory.
- AGU register N is written using `MOVEU.W`, zero extending in bits [23:16]. If N is used as an offset register in the `(Rj+N)` addressing mode, it must be sign extended to 24-bits. (Refer to Section 5.4.2 on page 5–5).
- When legacy and new instructions are mixed, for any AGU register loaded with a 56800 instruction and used in a 56800E instruction, it is imperative to use either `SXTA.W` or `ZXTA.W` to get the intended sign extension. Whenever an AGU is used as an offset, the upper 8 bits must be loaded with the sign value using `SXTA.W`. When an AGU is used as a pointer, the upper 8 bits must be zeroes. This can be done using `ZXTA.W`. Refer to Section 5.7 on page 5–9.
- When immediate data is written to AGU registers R0–R3 and N:
  - Use “`MOVE.W  #xx,HHHH`” to destination R0–R3, for values `xx` inside the range [0, 63].
  - Use “`MOVEU.W  #xxxx,SSSS`” to destination R0–R3, for values xxxx outside the range [0, 63].
  - Use “`MOVE.W  #xx,HHHH`” to destination N, for values of `xx` inside the range [-64, 63].
  - Use “`MOVEU.W  #xxxx,SSSS`” to destination N, for values of xxxx outside the range [-64, 63]. If N is used as an offset, follow this instruction with “`SXTA.W  N`” to preserve the intended 24-bit sign value.
- When immediate data is written to the HWS and LA registers:
  - Use “`MOVEU.W  #xxxx,SSSS`” for all values of `xxxx`

For register field definitions `HHHH` and `SSSS`, refer to Table A-5 on page A-3 and Table A-8 on page A-5.

4.2 Issues with AGU Arithmetic

NOTE:

It is very rare that a program is written in a manner where the AGU registers are expected to overflow (or underflow). If it is necessary to guarantee correctness even for this unusual coding style, then this section will be useful; otherwise, this section can be bypassed.

A compatibility issue can arise due to the 16-bit width of the DSP56800's AGU registers, address buses, and datapath. The problem occurs when an AGU computation overflows the DSP56800's highest possible address ($00FFFF) or underflows the lowest possible address ($000000). This can occur during an effective address calculation or address register update via one of the post-update addressing modes for any of the following DSP56800 instructions:

- Data ALU arithmetic or bit manipulation instructions with one operand in memory
- Move instructions
- Single and dual parallel move instructions
- `TSTW` (not mapped to the legacy 56800E instruction)

The problem also occurs for AGU calculations using the DSP56800's `LEA` instruction. Two examples of AGU calculations in Table 4-1 demonstrate this problem.

Table 4-1. Demonstrating DSP56800 AGU Overflow and Underflow

Example Calculation	DSP56800 Result (16-bit AGU Arithmetic)	DSP56800E Result (24-bit AGU Arithmetic)	Comments
$00FFFC + $000010	$00000C	$01000C	AGU Overflow
$000003 - $000007	$00FFFC	$FFFFFC	AGU Underflow

NOTE:

It is not good programming practice to write application code that depends on the wrapping effect of an overflow from 64K to 0 or an underflow from 0 to 64K (i.e., the natural modulo effect on 16-bit AGU registers). Nevertheless, most code written in this matter is covered by special legacy instructions that are defined to emulate the wrapping behavior from the DSP56800 architecture (see Section 5.1 on page 5–1). However, a few cases are not covered by these legacy instructions. Refer to Section 4.2.4 on page 4–5.

Four different techniques discussed in the following sections are used on the DSP56800E architecture to address this problem:

- Cases solved with legacy instructions when using linear addressing
- Cases solved by careful instruction definition
- Cases solved by adding a zero-extend instruction
- Cases not handled by special legacy instructions that can be broken into more than one instruction sequence

4.2.1 Cases Solved by Legacy Instructions—Linear Addressing

When linear addressing is used, address wrapping on the DSP56800 16-bit AGU registers R0–R3 is exactly reproduced on the DSP56800E by using the following special legacy instructions:

- `MOVE      X:(Rj+xxxx),DDDDD`
- `MOVE      DDDDD,X:(Rj+xxxx)`
- `MOVE      X:(Rj+N),DDDDD`
- `MOVE      DDDDD,X:(Rj+N)`
- `LEA       (Rj+xxxx)`
- `LEA       (Rj)+N`
- `TSTW      X:(Rj+xxxx)`
- `TSTW      X:(Rj+N)`

Consider the DSP56800 code shown in Code Example 4-1.

Code Example 4-1. Original DSP56800 Code

```
        MOVE            #$C000,R0    ; Load 16-bit pointer with $C000
        LEA             (R0+$8000)   ; Add $8000 to this value
```

The calculated effective address is $4000. When this code is directly executed on the DSP56800E architecture, it is executed as the two instructions in Code Example 4-2.

Code Example 4-2. Same Code Mapped to DSP56800E

```
        MOVEU.W         #$C000,R0    ; Load 24-bit pointer with $00C000
        LEA             (R0+$8000)   ; Add $8000 to this value
```

If the effective address were calculated using the normal 24-bit modulo arithmetic of the DSP56800E, the result would be $014000 rather than the original result of $4000. However, the calculated result is actually $004000, the same value generated on the DSP56800. This is because the legacy instructions `LEA`, `MOVE`, and `TSTW` utilize special addressing modes that perform 16-bit AGU arithmetic by zeroing the upper 8 bits of the 24-bit result. These special instructions are shown in the shaded areas in Table A-10 on page A-8.

NOTE:

The DSP56800E architecture does not support address wrapping for the SP register, which only uses 24-bit arithmetic. Thus, the legacy instructions do not apply to the stack pointer.

4.2.2 Cases Solved by Definition of Operation

In some cases, the definition of an operation helps ensure DSP56800 compatibility. In this case there is no compatibility issue.

Consider the DSP56800 code shown in Code Example 4-3.

Code Example 4-3. Original DSP56800 Code

```
MOVE    #$1004,R0    ; Load 16-bit pointer with $1004
MOVE    #$-4,X0      ; Load negative 16-bit value in X0 ($FFFC)
MOVE    X0,N         ; Load N with negative value from X0 reg
MOVE    X:(R0)+N,X0  ; Updates R0 with -4 ($FFFC)
```

On the DSP56800 architecture, the result in the R0 register after this code sequence is $1000 due to its 16-bit AGU unit. The '1' located in bit 17 on DSP56800E is not available on the DSP56800 architecture due to its 16-bit datapath. This is shown in Code Example 4-4.

Code Example 4-4. Correct Execution on DSP56800E Architecture

```
MOVEU.W #$1004,R0    ; Load 16-bit pointer with $001004
MOVEU.W #$-4,X0      ; Load negative value into X0 ($FFFC)
MOVEU.W X0,N         ; Loads N w/ positive value ($00FFFC)
                     ; (MOVEU.W instruction zero extends value)
MOVE.W  X:(R0)+N,X0  ; Updates R0 with $FFFFFC, not $00FFFC
                     ; (due to the defn of (Rn)+N addr mode)
```

On the DSP56800E architecture, there should be a problem because the 24-bit N register does not contain a negative value but instead contains a large positive value, $00FFFC). The result in the R0 register after this code sequence, however, is also $1000, identical to the result calculated on the DSP56800. The reason that the DSP56800 provides the correct answer is due to the fact that the `(Rn)+N` addressing mode is defined as a 16-bit address computation. This addressing mode ignores the upper 8 bits of the N register and sign extends from bit 15 before performing the addition. Thus, both architectures update R0 using the value $FFFFFC and both calculate identical results.

4.2.3 Cases Solved by Adding a Zero-Extend Instruction

In most cases, AGU overflow and underflow compatibility issues are directly handled by the assembler by mapping these instructions with their legacy counterparts in the DSP56800E instruction set.
Code Example 4-6 demonstrates DSP56800 code where this is the case.

Code Example 4-5. Correct DSP56800E Execution if no AGU overflow/underflow

```
MOVEU.W       #$F000,R0    ; Load 16-bit pointer with $F000
MOVEU.W       #$0004,N     ; Load $0004 into the N register
NOP

ADD  X0,A     X:(R0)+N,X0  ; R0 = $F000 + $0004 = $F004
                           ;   on the DSP56800E
                           ;   (no AGU overflow occurs)
```

Compatibility issues arise for the `X:(Rn)+N` addressing mode if it is used in a manner where it overflows the value $00FFFF or underflows the value $0000 boundary, such as in Code Example 4-6.

Code Example 4-6. Original DSP56800 Code

```
MOVE     #$F000,R0    ; Load 16-bit pointer with $F000
MOVE     #$2000,N     ; Load $2000 into the N register
NOP                   ;
ADD      X0,A   X:(R0)+N,X0; R0 = $F000 + $2000 = $1000 on DSP56800
                      ; (AGU overflow occurred — 16-bit ALU)
```

One solution to this problem is to correct the AGU register after the post update with the zero extension instruction `ZXTA.W`, as shown in Code Example 4-7.

Code Example 4-7. Correct Execution on DSP56800E Sequence

```
MOVEU.W #$F000,R0          ; Load 16-bit pointer with $F000
MOVEU.W #$2000,N           ; Load $2000 into the N register
NOP                        ;
ADD  X0,A    X:(R0)+N,X0   ; R0 = $00F000 + $002000 = $011000
ZXTA.W  R0                 ; R0 is corrected by zero extending
                           ;   the upper 8-bits on R0
```

Another solution is to break the original DSP56800 parallel instruction into separate DSP56800E instructions, as discussed in the next section. Breaking the instruction avoids the overflow (or underflow), while the zero-extension method corrects the result after the overflow.

The following addressing mode cases are not handled by special legacy instructions. They can also be corrected after the overflow (or underflow) takes place by adding the zero-extension instruction, "`ZXTA.W  Rk`" after the post-update of `Rk`.

- `(Rk)+`
- `(Rk)-`
- `(Rk)+N`

If it is known that there is no AGU overflow or underflow in an application and it is important to maintain peak performance, it is not necessary to zero extend or break the original DSP56800 instruction.

4.2.4 Cases Solved by Breaking Into More Than One Instruction

There are some cases of AGU overflow or underflow that are not solved by the legacy instructions or the definitions of the operations. The following addressing modes are subject to compatibility mismatch on overflow or underflow:

- Cases that can also be corrected by adding the zero-extension instruction, "`ZXTA.W  Rk`" (see Section 4.2.3 on page 4–4):
  - `(Rk)+`
  - `(Rk)-`
  - `(Rk)+N`
- Cases mapped to the DSP56800E standard `(Rk+xxxx)` addressing mode, (see Section 5.2 on page 5–2 for a complete list of instructions with this addressing mode):
  - `(R2+xx)`, when this addressing mode is mapped to the standard mode

Code Example 4-8 demonstrates addressing modes that are subject to compatibility mismatches when the DSP56800's 16-bit AGU register width overflows or underflows.

Code Example 4-8. Original DSP56800 Code

```
        MOVE    #$FFFF,R2          ; Load 16-bit pointer with $FFFF
        MOVE    #$2000,X:(R2+3)    ; Write $2000 to location $0002
        NOP                        ;
        BFSET   #$FF00,X:(R2+5)    ; Set $FF00 on location $0004
```

When the user does not correct this sequence, overflow occurs as shown in Code Example 4-9.

Code Example 4-9. Original Code Mapped to DSP56800E Syntax

```
        MOVEU.W #$FFFF,R2      ; Load 16-bit pointer with $00FFFF
        MOVE.W  #$2000,X:(R2+3) ; Write $2000 to location $010002
                       ; (AGU overflow occurs — 16-bit ALU)
        NOP                    ;
        BFSET   #$FF00,X:(R2+5) ; Set $FF00 on location $010004
                       ; (AGU overflow occurs — 16-bit ALU)
```

By partitioning the instruction into two parts, the AGU register can be safely post-updated to avoid the overflow by using the `LEA` instruction. This is shown in Code Example 4-10.

Code Example 4-10. Breaking Up the Original Sequence To Correct Overflow

```
        MOVE    #$FFFF,R2    ; Load 16-bit pointer with $FFFF
;NEW SEQUENCE
        LEA     (R2+3)       ; Update R2 with value $0002
        MOVE    #$2000,X:(R2) ; Write $2000 to location $0002
        LEA     (R2-3)       ; Update R2 with value $FFFF

        NOP                  ;

;NEW SEQUENCE
        LEA     (R2+5)       ; Update R2 with value $0004
        BFSET   #$FF00,X:(R2) ; Set $FF00 on location $0004
        LEA     (R2-5)       ; Update R2 with value $FFFF
```

The `(SP–xx)` addressing mode is also subject to mismatch on overflow or underflow. The DSP56800E architecture does not support wrapping the stack pointer at the $FFFF boundary. Refer to Section 5.4.2 on page 5–5.

Chapter 5
Compatibility Issues

This section outlines potential assembly language incompatibilities between the DSP56800 and the DSP56800E and provides reference to other places in the document where these cases are discussed in detail. The assembler is designed to select the correct mapping and resolve most of the issues that arise when porting DSP56800 assembly code to the DSP56800E architecture. In other cases, the coding style used on legacy code can affect this process and may require user to map the instruction (or sequence of instructions) manually to ensure correct execution. This document currently covers assembly language issues only; a future revision will also include a section on issues pertaining to porting C-language programs from the DSP56800 to the DSP56800E architecture.

5.1 New Special Legacy Instructions

Two new addressing modes are provided for the expressed purpose of matching the 16-bit AGU register widths exactly with key addressing modes found on the DSP56800:

• `X:(Rj+N)`	Indexed by Offset Register N—Legacy Version
• `X:(Rj+xxxx)`	Indexed by 16-Bit Displacement—Legacy Version

These addressing modes share a common trait—the upper 8-bits of the address value are forced to zeroes. This refers to the effective address computed by the instruction.

NOTE:

It is recommended that the addressing modes in this class not be used in new applications written for the DSP56800E.

This section lists the instructions that make use of these addressing modes. These instructions are found only in this guide and are highlighted in Table A-10 on page A-8. Details on these instructions can be found in section Section A.3, “Legacy Instruction Summary Tables,” on page A-24. With no exceptions, the following list is only applicable for AGU registers R0, R1, R2 and R3. In cases where the original instruction uses these two modes with the stack pointer, SP, the mapping table utilizes the standard DSP56800E translation. To illustrate this, Code Example 5-1 shows how “`LEA (R0)+N`” and “`LEA (SP)+N`” are mapped.

Code Example 5-1. Original DSP56800 Code

```
;Demonstrating Legacy Mapping on LEA for R0 and SP

8420     LEA    (R0)+N       ; mapped to LEA (R0)+N

89BB     LEA    (SP)+N       ; mapped to ADDA N,SP

89BB     ADDA   N,SP
```

The following instructions, which appear only in this guide, are characterized by the unique way in which address arithmetic is performed—the upper 8-bits of the 24-bit AGU register are forced to zeroes to form the effective address of the instruction.

• `LEA (Rj)+N`	– update `Rj` by `N` (`N` treated as a 16-bit register)
• `LEA (Rj+xxxx)`	– update `Rj` by `#xxxx`
• `MOVE X:(Rj+xxxx),DDDDD`	– read value from memory location `Rj+xxxx`
• `MOVE X:(Rj+N),DDDDD`	– read value from memory location `Rj+N`
• `MOVE DDDDD,X:(Rj+xxxx)`	– write reg value to memory location `Rj+xxxx`
• `MOVE DDDDD,X:(Rj+N)`	– write reg value to memory location `Rj+N`
• `TSTW X:(Rj+xxxx)`	– test value from memory location `Rj+xxxx`
• `TSTW X:(Rj+N)`	– test value from memory location `Rj+N`

5.2 Replacement of (R2+xx) with (Rj+xxxx)

The `(R2+xx)` addressing mode was included in the DSP56800 to provide an opcode with optimal size for an indirect addressing mode using a 6-bit positive offset. This addressing mode has been removed from the DSP56800E architecture to preserve opcode space. Instructions using this addressing mode are mapped by the DSP56800E assembler to instructions using one of the following two DSP56800E addressing modes:

- Standard `(Rn+xxxx)` Address Mode. This addressing mode uses 24-bit arithmetic to compute the effective address, and requires an additional word for the final opcode. This is true for the following instructions:
  - `BFCHG    #xxxx,X:(R2+xx)`
  - `BFCLR    #xxxx,X:(R2+xx)`
  - `BFSET    #xxxx,X:(R2+xx)`
  - `BFTSTH   #xxxx,X:(R2+xx)`
  - `BFTSTL   #xxxx,X:(R2+xx)`
  - `BRCLR    #xxxx,X:(R2+xx),AA`
  - `BRSET    #xxxx,X:(R2+xx),AA`
  - `MOVE     #xxxx,X:(R2+xx)`

- Legacy `(Rn+xxxx)` Address Mode (see Section 5.1.) This addressing mode uses 16-bit arithmetic to compute the effective address, and also requires an additional word for the final opcode. This is true for the following instructions:
  - `TSTW     X:(R2+xx)`
  - `LEA      (R2+xx)`
  - `MOVE     X:(R2+xx),8-HHHH`
  - `MOVE     8-HHHH,X:(R2+xx)`

The mapping of all the above cases generates an additional program opcode word in the 56800E instruction syntax. Note that wherever code growth is generated, some target labels in change-of-flow instructions may not be reachable, resulting in a link error (due to the unresolved reference). This affects only those change-of-flow instructions which utilize a signed, 7-bit, PC-relative offset to represent the location of the target label from the program counter. When segments of code increase in program size, the location of these references may fall beyond the 7-bit offset boundary and thus become unresolved.

Another instance where growth in code size affects an instruction sequence occurs when the REP instruction is followed by one of these instructions. Instructions that produce code growth are highlighted in Section 5.4 on page 5–4.

5.3 Compatibility Issue with Modulo Addressing

The following list of special legacy instructions using `(Ri+xxxx)` for Ri = {R0 or R1} may not behave correctly when *modulo addressing* is active. If modulo addressing is active and offset `xxxx` is negative, the CW Assembler generates a warning message indicating that the instruction may operate incorrectly (for R0 and R1 only).

- `MOVE     X:(Ri+xxxx),DDDDD     (R0 or R1)`
- `MOVE     DDDDD,X:(Ri+xxxx)     (R0 or R1)`
- `TSTW     X:(Ri+xxxx)           (R0 or R1)`

The user should select a mapping which provides exact behavior. The selection of the mapping has dependency on the interpretation for the immediate value, `xxxx`. Two possible interpretations are possible for `xxxx`:

- “offset value” represented by the signed value in `xxxx`,
- “base address pointer” represented by the lower 16-bits in the R0 or R1 register.

The first interpretation, where the value of xxxx represents the offset, is more typical. In this case, the user must manually modify the legacy instructions listed above using the following syntax:

- `MOVE.W   X:(Ri+ Sxt32:xxxx),DDDDD     (R0 or R1)`
- `MOVE.W   DDDDD,X:(Ri+ Sxt32:xxxx)     (R0 or R1)`
- `TST.W    X:(Ri+ Sxt32:xxxx)           (R0 or R1)`

In these instructions, `Sxt32:xxxx` defines a 32-bit sign extended value. When the value `xxxx` is a negative offset and is represented as a hexadecimal number, the sign must be extended to 32 bits, e.g., $FFF8 should be replaced by $FFFF FFF8.

When `xxxx` is interpreted as the base address of a memory location, the value of `xxxx` must remain a 16-bit unsigned value, as is the case for every AGU register representing a pointer. In this case, the AGU register represents the offset and must be sign extended to 24 bits with "`SXTA.W {R0 or R1}`" (see Section 4.1, "Initializing AGU Registers," on page 4-1). In order to remain compatible, the following sequence must be used for each affected instruction:

- `For MOVE.W X:(R0+xxxx),DDDDD` (or using `R1`)

```
SXTA.W    R0                 – sign extend R0 to 32-bits
LEA       (R0+xxxx)          – adjust R0 using modulo arithmetic
MOVE      X:(R0),DDDDD       – move data from memory to register
LEA       (R0-xxxx)          – restore R0 to its original value
```

- `For MOVE.W DDDDD,X:(R0+xxxx)` (or using `R1`)

```
SXTA.W    R0                 – sign extend R0 to 32-bits
LEA       (R0+xxxx)          – adjust R0 using modulo arithmetic
MOVE      DDDDD,X:(R0)       – move data from register to memory
LEA       (R0-xxxx)          – restore R0 to its original value
```

- `For TSTW X:(R0+xxxx) (or using R1)`

```
SXTA.W    R0                 – sign extend R0 to 32-bits
LEA       (R0+xxxx)          – adjust R0 using modulo arithmetic
TSTW      X:(R0)             – test data in memory location
LEA       (R0-xxxx)          – restore R0 to its original value
```

5.4 Instructions That Produce Code Growth

After assembling legacy code, the user may observe that code size has increased. There is no guarantee that code size will remain the same after legacy code is assembled with the DSP56800E assembler. The source of code growth can be divided into five main categories. Code growth can be observed as a result of the following factors:

1. Pipeline dependencies
2. Sign requirements for the N register
3. Change-of-flow instructions
4. Hardware loops
5. Automatic mappings requiring an extra word

5.4.1 Pipeline Dependencies

Code growth can result from automatic insertions of NOPs (software stalls) by the DSP56800E assembler due to pipeline dependencies.

In Code Example 5-2 and Code Example 5-3, there is a pipeline dependency that occurs when the R0 register is copied to R1 by the Tcc instruction and the immediately following instruction modifies the R0 register contents in an addressing mode or in an AGU calculation.

Code Example 5-2. Dependency after a Tcc Instruction with R0 Modification

```
TEQ           A,B    R0,R1  ; R0 copied to R1 by Tcc instruction
MOVE.W        X:(R0)+,Y1    ; R0 contents modified in AGU calculation
```

Code Example 5-3. Dependency after a Tcc Instruction with R0 Modification

```
TEQ           A,B    R0,R1  ; R0 copied to R1 by Tcc instruction
ADDA          R2,R0         ; R0 contents modified in AGU calculation
```

If either of these instruction sequences is detected, the assembler generates a warning and inserts one NOP between the two instructions as shown in Example 1-14 and 1-15.

Code Example 5-4. Assembling Sequence with AGU Dependency

```
TEQ           A,B    R0,R1  ; R0 copied to R1 by Tcc instruction
NOP                         ; Assembler inserts NOP - removes dependency
MOVE.W        X:(R0)+,Y1    ; R0 contents modified in AGU calculation
```

Code Example 5-5. Assembling Sequence with AGU Dependency

```
TEQ           A,B    R0,R1  ; R0 copied to R1 by Tcc instruction
NOP                         ; Assembler inserts NOP - removes dependency
ADDA          R2,R0         ; R0 contents modified in AGU calculation
```

Due to the DSP56800E pipeline, the value moved to R1 by the first instruction in the last two examples is the value of R0 *after* it is updated by the second instruction in the sequence. This behavior is consistent even with interrupts, primarily because interrupts are not permitted after a Tcc with an AGU transfer and the instruction immediately following it. Refer to the section titled “Non-Interruptible Instruction Sequences” in the *DSP56800E Core Reference Manual*.

5.4.2 24-bit Signed Requirement for the N Register

There are several cases where N is required to be a 24-bit sign extended offset register. This is achieved by preceding the following instructions with “`SXTA.W  N`”, which correctly sign extends the N register to 24-bits (refer to Section 5.7 on page 5–9).

- `LEA      (SP)+N`
- `MOVE     X:(SP+N),8-HHHHH`
- `MOVE     X:(SP+N),8-SSSS`
- `MOVE     8-DDDDD,X:(SP+N)`
- `TSTW     X:(SP+N)`

5.4.3 Extending the Reach on Change-of-Flow Instructions

Localized code growth may place target labels outside the reach of the linker and generate link errors. User intervention is required to clear errors resulting from localized code growth. These cases occur when target labels become unresolved references because they fall beyond the 7-bit offset boundary during the porting process to the DSP56800E. They occur only in the following four instructions:

- `Bcc      <OFFSET7>`
- `BRA      <OFFSET7>`
- `BRCLR    #MASK8,<source>,<OFFSET7>`
- `BRSET    #MASK8,<source>,<OFFSET7>`

When code growth has produced cases where labels fall beyond the 7-bit offset, the forcing operator “>” must be applied to the operand for the `BCC` and `BRA` instructions to force the assembler to use `<OFFSET18>` instead of `<OFFSET7>`. When the `BRCLR` or `BRSET` instruction is responsible for the unresolved reference, the user must break up the instruction in the following manner:

- For `BRCLR #MASK8,<source>,<OFFSET7>` use:

```
BFTSTL        #MASK16,<source>
BCS           ><OFFSET18>       – note use of forcing operator “>”
```

- For `BRSET #MASK8,<source>,<OFFSET7>` use:

```
BFTSTH        #MASK16,<source>
BCS           ><OFFSET18>       – note use of forcing operator “>”
```

In the latter two cases, code growth is increased by two words.

5.4.4 Adding a NOP to a Hardware Loop

The hardware do loops `DO` and `REP` must be mapped to the `DOSLC` instruction when the LC register is used as an operand. The `DO` instruction is automatically mapped by the assembler, while the `REP` instruction requires the user to complete the translation and add the target label (last address). This occurrence is uncommon and limited to the following two cases:

- `DO    LC,xxxx` – where `xxxx` represents a 16-bit absolute address

  The assembler maps the `DO` instruction to the new instruction, `DOSLC`. If the body of the original loop is 1 word in length, the assembler inserts a single `NOP` instruction in the loop body to complete the 2-word minimum requirement for the `DOSLC` instruction.

- `REP   LC` – The next single word instruction is repeated LC times

  When the `REP` instruction uses `LC` as the operand, the assembler flags an error message. The user must map this case to the `DOSLC` instruction. A `NOP` instruction must be added to the body of the loop to satisfy the 2-word minimum requirement, and a label must be added to complete the structure of the loop instruction. Note that unlike `REP`, the `DOSLC` instruction can be interrupted.

Restrictions on hardware loops are described in more detail in Section 5.10 on page 5–11.

5.4.5 Automatic Mappings Requiring an Extra Word

The following instructions are automatically recognized and mapped by the assembler using an additional word in the new opcode:

- `ADD` `X:aa,FDD` – add value in short address loc `X:aa` to reg
- `SUB` `X:aa,FDD` – subtract value in short address loc `X:aa` to reg
- `CMP` `X:aa,FDD` – compare value in short address loc `X:aa` to reg
- `DECW` `X:aa` – decr value in short address location `X:aa`
- `INCW` `X:aa` – incr value in short address location `X:aa`
- `MOVE` `X:(SP-xx),F1` – read stack indexed by `xx:[1,64]` to {`A1` or `B1`}
- `MOVE` `F,X:(SP-xx)` – move {`A` or `B`} to stack indexed by `xx:[1,64]`
- `MOVE` `#xx,Rj` – move signed 7-bit integer to AGU register
- `MOVE` `#xx,F1` – move signed 7-bit integer to {`A1` or `B1`}
- All instructions using the `(R2+xx)` addressing mode (see Section 5.2 on page 5–2).

5.5 Changes Related to the I/O Short Addressing Mode

Another compatibility issue to consider is the `X:<<pp` addressing mode. In the current DSP56800E implementation, the I/O short address (or peripheral address) is generated by concatenating two objects—an 18-bit value (obtained from 18 input terminals to the core), and a 6-bit value extracted from the instruction opcode. This concatenation forms the complete 24-bit data address reference. On the DSP56800, the peripheral portion of the memory map is located between addresses X:$FFC0 and X:$FFFF. The following notations are accepted by the DSP56800 assembler as one-word instructions using this addressing mode:

- `MOVE` `X:$FFC1,X0` – upper 10 bits hardwired to $3FF
- `MOVEP` `X:$FFC1,X0` – upper 10 bits hardwired to $3FF
- `MOVE` `X:<<$FFC1,X0` – uses I/O short forcing operator
- `MOVEP` `X:<<$FFC1,X0` – uses I/O short forcing operator

For exact DSP56800 compatibility, the value on the 18-bit input bus must be $003FF when the processor exits the Reset processing state. This ensures that all instructions accessing the peripheral space access the same physical location in peripheral memory ranging from location $00FFC0 to $00FFFF. The user must be aware of the rules described in Section 4.1 on page 4–1 when initializing pointer registers, especially when these registers are used with indirect addressing modes to access the peripheral space.

NOTE:

The CodeWarrior IDE has a switch that allows the assembler to accept both DSP56800 and DSP56800E syntax in the target code. It also configures the data memory map to reside within 64K of memory space. In order to provide added flexibility in legacy applications, two exceptions are allowed—the peripheral space and the system stack space. These areas can reside anywhere within the 24-bit range of data memory space.

5.6 Strategy for Loading AGU Registers

This section describes the strategy followed when AGU registers used by the toolset are loaded during DSP56800 program assembly. This information can also be useful when mixing instructions from both architectures.

For exact operation of existing DSP56800 code, the user must be aware of some basic rules when writing to the AGU or Program Controller registers. In the DSP56800, these registers are 16 bits, but are 24 bits in the DSP56800E. When legacy code writes a value to SP, R0–R3, N, HWS or LA, it must be guaranteed that the code still runs correctly on the DSP56800E.

To prevent compatibility problems with legacy code during the loading of pointer values, the DSP56800E provides several unsigned word load instructions to the AGU pointer registers. When AGU registers are loaded with an offset, the 24-bit signed value must be preserved.

The following examples illustrate potential compatibility issues that the user must be aware of in order to avoid incorrect behavior in the DSP56800E. These cases are automatically handled by the assembler, but the user must be aware of these issues in order to avoid hidden problems due to coding style.

- `MOVE #xx,Rj` – initialize AGU registers with a 7-bit short value

When this instruction is used to initialize R0, R1, R2 or R3 with a negative short immediate value (–1 to –64), and the register is used as a pointer to represent an address ($FFFF to $FFC0), the unsigned move instruction must be used to ensure that the upper 8-bits of the address be zero. In the DSP56800E this operation is performed with a `MOVEU.W` instruction to ensure that the upper 8-bits are zero. This case is illustrated in Code Example 5-6.

Code Example 5-6. Loading Pointer with Immediate Data

```
        MOVE    #-1,R0        ; Load 16-bit pointer with ($00FFFF)
                              ; (1 word DSP56800 instruction)
        NOP                   ; Assembler maps to 2 word MOVEU.W  #-1,R0
        MOVE    X:(R0),X0     ; Move value from X:$00FFFF to X0
```

- `MOVE #xxxx,Rj` – initialize AGU reg with 16-bit immediate values

The R0–R3, HWS, and LA registers on the DSP56800 are used directly or indirectly for addressing memory. In order to maintain compatibility, it is necessary that 16-bit values from legacy programs be written to these registers with zero extension. This ensures that the address values stored in these registers access memory locations within the first 64K of the memory map. This is not always the case for the N register, as described in Section 5.7 on page 5–9. In Code Example 5-7, the incorrect value $FF9001 would be loaded in R0 were it not for the `MOVEU.W` instruction that automatically replaces the `MOVE`.

Code Example 5-7. Loading Pointer with 16-bit Value

```
      MOVE      #$9001,R0     ; Load $009001 to R0
                              ; Assembler maps to MOVEU.W  #$9001,R0
```

- `MOVE Y1,Rj` – initialize AGU reg from other registers or from memory

Loading values to AGU registers from other registers or from memory must also be written with zero extension. This ensures that the values stored in these registers represent the same absolute magnitude intended in the original code, and maintain the integrity of the original program. This case is illustrated in Code Example 5-8.

Code Example 5-8. Loading Pointer from Register or Memory

```
        MOVE    X:$C300,R0    ; Load value in X:$00C300 to R0
                              ; Assembler maps to MOVEU.W  X:$C300,R0
        MOVE    Y1,R1         ; Move value in Y1 to R1
                              ; Assembler maps to MOVEU.W  Y1,R1
```

5.7 Requirements for {R0-R3, N}

5.7.1 As Pointer Registers

Registers R0–R3 are most commonly used to represent pointer values. Because the DSP56800 memory model is limited to 64K words, these registers are always initialized with the mapped instruction `MOVEU.W` to ensure that the upper 8-bits of these registers contain zeroes. Instructions that initialize the N register are commonly mapped to `MOVEU.W` with one exception. When N is initialized with an immediate number within the range [-64,-63], `MOVE.W` is used instead of the unsigned move to sign extend N to 24 bits.

When R0–R3 or N are used to represent a base address in mixed mode with instructions defined in the DSP56800E instruction set, the upper 8-bits must be zeroes to guarantee compatibility with the 56800.

5.7.2 As Offset Registers

The N register is most commonly used as an offset register. When registers R0–R3 and N are used as offset registers, it is important that the upper 8-bits contain signed information. This is done automatically only when the register N is initialized with an immediate number within the range [-64,-63], in which case a `MOVE.W` instruction is used.

Correct sign extension is specifically important when the AGU registers are used in mixed mode with instructions defined in the DSP56800E instruction set, as illustrated in Code Example 5-9. In these cases the instruction using these registers must be preceded by the AGU sign extend instruction `SXTA.W` to guarantee that the upper 8 bits contain sign information and the 24-bit arithmetic correctly reproduces the same effective address as the 16-bit computation.

Code Example 5-9. Mixed Instructions & Loading Pointers w/ Offset Value

```
MOVE    #$8000,R0          ; Load 16-bit ($008000)
MOVE    #-1,X0             ; Load 16-bit ($FFFF)
MOVE    X0,N               ; Mapped to MOVEU.W ($00FFFF)
MOVE    X0,R1              ; Mapped to MOVEU.W ($00FFFF)

SXTA.W  N                  ; Sign Extend to 24-bits ($FFFFFF)
                           ; (mixing from DSP56800E instr. set)
                           ; If N not sign extended, the wrong
                           ; location, X:017FFF, is used
MOVE.W  X:(R0+N),X0        ; Move value from X:$007FFF to X0

SXTA.W  R1                 ; Sign Extend to 24-bits ($FFFFFF)
                           ; (mixing from DSP56800E instr. set)
ADDA    R1,R0              ; Move value from X:$007FFF to X0
MOVE.W  X:(R1+BASE),X0     ; R1 acting as an offset
```

5.8 Reading the LC Register

The LC register in the DSP56800 is only 13 bits, but has been expanded to 16 bits in the DSP56800E. Bits 15:13 of the DSP56800 LC register are always read as zeroes, whereas on DSP56800E all 16 bits of the register are read. There may be an unusual case where the user's original intention was to strip the upper three bits of a 16 bit value. Consider the case where a given value has dual use—the upper 3-bits and lower 13-bits are packed into one 16-bit value. In the DSP56800 architecture, the user can take advantage of the width of LC to strip the upper 3 bits and use the resulting 13-bit value to initialize a register, as shown in Code Example 5-10.

Code Example 5-10. Copying a 13-bit Value from the LC Register—DSP56800

```
MOVE    #$C1FF,X0    ; X0 set to $C1FF - Upper 3 bits are not 0
MOVE    X0,LC        ; Info in upper 3 bits lost, 13-bit LC
MOVE    LC,N         ; N loaded: $01FF - Information stripped
```

If exact compatibility in this situation is required, the alternative shown in Code Example 5-11 solves this problem. The solution is simply to clear the upper 3 bits of the LC register using an `ANDC` instruction alias whenever it is written from a register or memory location.

Code Example 5-11. Copying a 13-bit Value from the LC Register—DSP56800E

```
MOVE.W  #$C1FF,X0    ; X0 set to $C1FF - Upper 3 bits are not 0
MOVE.W  X0,LC        ; Info in upper 3 bits not lost
ANDC    #$1FFF,LC    ; <=== Clear upper 3 bits of LC reg ******
MOVEU.W LC,N         ; N loaded with $01FF
```

Now on both architectures, the value loaded into N is $01FF. This technique can be used for both `REP` and `DO` loops.

5.9 Numeric Target References

The DSP56800 assembler accepts the use of numeric values for targets instead of the traditional labels in change-of-flow instructions (`JCC, JMP, JSR, BCC, BRA, BRCLR, BRSET and DO`). However, the DSP56800E assembler can only accept targets defined as traditional labels, and rejects all numeric references in change-of-flow instructions. This is because of the many factors that can lead to code growth (see Section 5.4 on page 5–4), particularly the insertion of `NOP`s to remove pipeline dependencies.

These cases and examples are as follows:

- `<Jcc, JMP, JSR>    <Numeric_Absolute_Address>`
  - `JMP    $2000` – can't rely on numerical absolute target
  - `JSR    AbsValue` – where `AbsValue` is defined as a numeric value
- `<Bcc, BRA>    <Numeric_Relative_Offset>`
  - `Bne    *-7` – can't rely on numerical offset
  - `Beq    $C100` – can't rely on absolute target
  - `BRA    *+Value` – can't rely on numerical offset
- `<BRCLR, BRSET>    #MASK8,<source>,<Numeric_Relative_Offset>`
  - `BRSET    #001F,A,*-10` – can't rely on numerical offset
- `DO    <source>,<Numeric_Absolute_Address>`
  - `DO    #3,$1000` – the body of the loop may have grown

Another compatibility issue can arise when a `JSR` instruction is executed, even when a proper label is used. When the target of a `JSR` is a different program address than in the original code, the return address will also be different. This can lead to a compatibility problem in the unusual case where the return address is used (or examined), and the exact value is expected based on the original location of the `JSR` instruction.

5.10 Hardware Loop Restrictions

Hardware loops can be implemented in the DSP56800 architecture with two instructions, `REP` and `DO`. The following sections illustrate the ways in which hardware loops are affected during porting.

5.10.1 Restrictions Common to DO and REP

- Using the LC register as the loop count is not allowed. The DSP56800E assembler remaps the DO instruction a DOSLC loop. The user must manually map the REP to a DOSLC and insert the target (last address) label. Note that the REP instruction is not interruptible, while DOSLC is interruptible. A `NOP` instruction can be inserted in the loop body to complete the 2-word minimum requirement of the `DOSLC` instruction.
- Using the LA register as the loop count is not allowed. This case is not supported in the DSP56800E architecture and an error is generated.
- Using a register operand as the loop count with an unsigned value greater than $1FFF results in a different value loaded to the LC register in the DSP56800E architecture because the LC is register 16 bits, whereas in the DSP56800 it is 13-bits. (Refer to section Section 5.8 on page 5–10 for more information.)
- Using an extension register (A2 or B2) as the loop count with a negative value results in a different value loaded to the LC register in the DSP56800E architecture because after the 4 bits are sign extended to 16-bits and loaded to the LC, only 13-bits are used in the DSP56800. In the DSP56800E architecture, no truncation is performed.

5.10.2 Restrictions Specific To REP

- Only single word instruction can follow the `REP` instruction
- Accessing the LC register is not permitted inside a repeat loop. A `DO` or `DOSLC` loop must be used instead.
- Writing to the M01 register is not permitted inside a repeat loop.
- The `ENDDO` instruction is not permitted inside a repeat loop.
- A 1-word, 3-operand instruction in which a portion of the destination register is also used as a source register is not allowed inside a repeat loop.

The `REP` instruction is affected when it is followed by an instruction that exhibits growth in opcode. The `REP` instruction can only be followed by one-word instructions. Section 5.4 on page 5–4 lists all instructions that generate code growth. Some of these instructions can be assembled with a warning and others are rejected.

- The following instructions are assembled with a warning and a NOP is inserted in the body of the loop:
  - `TSTW X:(R2+xx)`
  - `MOVE X:(R2+xx),8-HHHH`
  - `MOVE 8-HHHH,X:(R2+xx)`
  - `MOVE X:(SP-xx),F1`
  - `MOVE F,X:(SP-xx)`
  - `MOVE #xx,Rj`
  - `MOVE #xx,F1`
- The following instructions are rejected with an error because the sequence cannot not be assembled using a REP instruction. These cases are considered unusual.
  - `ADD X:aa,FDD`
  - `SUB X:aa,FDD`
  - `CMP X:aa,FDD`
  - `DECW X:aa`
  - `INCW X:aa`
  - `LEA (R2+xx)`

5.10.3 Restrictions Specific To The DO Instruction

- Accesses to SR, OMR and LC registers are not permitted in the last address, LA, and LA-1 of the DO loop.
- Computation of the condition codes for ENDDO is different between the two architectures. In the DSP56800E, condition codes are not modified and in the DSP56800, N, Z, V and C are updated.
- Specification of a register as the loop count with value zero, will execute the loop 2^{13} times in the DSP56800. In the DSP56800E, execution begins with the instruction immediately after the body of the loop.
- On nested loops, reading either LA or LC register after the inner loop

5.11 Differences when Saturation is Enabled

There is a difference in the behavior between the two architectures when saturation is enabled (the SA bit in the OMR is set). The two architectures operate identically when the SA bit is not set.

Saturation can occur on the DSP56800 for the following instructions when SA bit was set:

- ADC
- SBC
- DIV
- IMPY16 (or IMPY)

On DSP56800E, these instructions do not saturate their results even if SA is set.

NOTE:

The DSP56824 manual states that saturation is not enabled for the `IMPY16` instruction. The chip, however, does have saturation enabled for this instruction.

For DSP56800 applications where SA is set, the saturating nature of `ADC`, `SBC`, and `DIV` can be achieved on the DSP56800E as shown in Code Example 5-12, 13, and 14 respectively.

Code Example 5-12. Emulating Saturation with ADC Instruction

```
        ADC     Y,F                 ; ADC performed ignoring SA on 56800E
        BRCLR   #$0010,SR,OVR       ; check if SA == 1
        SAT     F                   ; saturate if SA == 1
OVR
```

Code Example 5-13. Emulating Saturation with SBC Instruction

```
        SBC     Y,F                 ; SBC performed ignoring SA on 56800E
        BRCLR   #$0010,SR,OVR       ; check if SA == 1
        SAT     F                   ; saturate if SA == 1
OVR
```

Code Example 5-14. Emulating Saturation with DIV Instruction

```
        DIV     Y0,F                ; DIV performed ignoring SA on 56800E
        BRCLR   #$0010,SR,OVR       ; check if SA == 1
        SAT     F                   ; saturate if SA == 1
OVR
```

Note that these cases are not exact emulations because condition codes are set differently when saturation truly does occur in any of the above sequences. Another case which cannot be exactly emulated is illustrated in Code Example 5-15.

Code Example 5-15. Difficult Case—Repeat Looping with SA set

```
;Difficult DSP56800 Code Sequence (assumes SA is set)
        REP     #3                  ; Repeat next instruction 3 times
        DIV     X0,A                ; performed ignoring SA on DSP56800E
```

Exact replication of DSP56800 operation requires that a SAT instruction is executed immediately after each `DIV` instruction. This is not possible, however, because `REP` can only be performed on a single instruction, not on a pair of instructions. A hardware `DO` loop is required instead. This is also true if an `ADC` or `SBC` instruction is used within a repeat loop.

5.12 Computing the Zero Condition Code

There is a difference in the manner in which the zero condition code bit, Z, is calculated when the CC bit in the OMR (called the “CM” bit in the DSP56800E) is set for the following instructions:

- `INCW (or INC)`
- `DECW (or DEC)`

The difference occurs when the CC bit is set and the operand is an accumulator. On the DSP56800, the Z bit is calculated using the lowest 32 bits of the accumulator. On DSP56800E, the Z bit is calculated using the 16 bits in the MSP portion of the accumulator, bits [31:16].

NOTE:

This is only is an issue for DSP56800 applications where the CC bit is set. The two architectures operate identically when the CC bit is cleared.

5.13 Computing the Carry Condition Code

There is a difference in the manner in which the carry condition code bit, C, is calculated for the following instructions:

- `ADD <reg>,X:xxxx`
- `ADD <reg>,X:(SP-xx)`

The difference occurs in the bit location where the carry is detected. On the DSP56800, the C bit is calculated from bit 35 of the result. On DSP56800E, the C bit is calculated from bit 31 of the result.

This difference applies regardless of the values of the OMR’s SA or CM bits.

5.14 New Requirements for X/P Mode

A specific set of instructions must be executed both to enter X/P mode (i.e., execute instructions from data memory rather than program memory) and exit X/P mode (return to executing instructions from program memory.

In the code sequences presented in this section, it is very important that the instruction segment between setting or clearing the XP bit and the `JMP` instruction should not be single-stepped due to the sensitive nature of these operations. (These code sequences also appear in the “Program Controller” chapter in the *DSP56800E Core Reference Manual*.)

5.14.1 Entering X/P Mode

To enter X/P mode, the following sequence of operations must be performed:

1. Download the desired program—including interrupt vectors, interrupt service routines, and data constants—into data memory.
2. Disable interrupts in the status register (SR).
3. Set the XP bit in the operating mode register (OMR).
4. Jump to the first instruction in data memory.
5. Re-enable interrupts from code in data memory (if required).

The code sequence to implement these steps varies slightly depending on the size of the target address specified in the JMP to instructions in data memory. Code Example 5-16 shows the code for a 19-bit target address.

Code Example 5-16. Entering Data-Memory Execution Mode, 19-Bit Target Address

```
BEGIN_X   EQU    $1000         ; Beginning address of program in data memory

          ORG    P:            ; (indicates code located in program memory)
          .
          .
; Exact Sequence for Steps 3 through 5
          BFSET #$0300,SR      ; Disable Interrupts
          NOP                  ; (wait for interrupts to be disabled)
          NOP                  ; (wait for interrupts to be disabled)
          NOP                  ; (wait for interrupts to be disabled)
          NOP                  ; (wait for interrupts to be disabled)
          NOP                  ; (wait for interrupts to be disabled)
          BFSET #$0080,OMR     ; Enable data memory instruction fetches
          NOP                  ; (wait for mode to switch)
          NOP                  ; (wait for mode to switch)
; NOTE: Must Use Assembler Forcing Operator - Forces 19-bit Address
          JMP    >XMEM_TARGET        ; Jump to 1st instruction in data memory
          NOP                  ; (fetched but not executed)
          NOP                  ; (fetched but not executed)
          NOP                  ; (fetched but not executed)

          ORG P:BEGIN_X,X:BEGIN_X        ; (both must be the same value)
XMEM_TARGET
          ; Remember to RE-Enable Interrupts
```

If a 21-bit target address is specified, the code sequence differs slightly as follows:

- One NOP instruction only (rather than two) must be inserted between the BFSET instruction that sets the XP bit and the JMP instruction
- The '>>' assembler forcing operator (rather than '>') is specified in the JMP instruction.

Regular interrupt processing is supported in data-memory execution mode. The interrupt vector table and all interrupt service routines must be copied to data memory because program memory is completely disabled when data-memory execution mode is active. It is only necessary to provide the particular interrupt vectors and service routines for interrupts that actually occur during data memory execution.

During the transition in and out of data-memory execution mode, interrupts must be disabled.

The following restrictions apply when programs are executed from data memory:

- Instructions that perform two reads from data memory are not permitted.
- Instructions that access program memory are not permitted.
- Interrupts must be disabled when data-memory execution mode is entered or exited.

Instructions that perform one parallel move operation are allowed in this mode.

NOTE:

The code that is used to enter data-memory execution mode must contain the exact number of NOP instructions shown in Code Example 5-16. There can be *no* jumps or branches to instructions within this sequence.

5.14.2 Exiting X/P Mode

To exit X/P mode and switch back to executing instructions program memory, the following sequence of operations must be performed:

1. Disable interrupts in the status register.
2. Clear the XP bit in the operating mode register.
3. Jump to the return location in the program memory space.
4. Re-enable interrupts from code that is located in program memory space.

For a 19-bit target address, the code sequence given in Code Example 5-17 must be used to exit data-memory execution mode. For a 21-bit target address, the code differs in similar fashion to entering X/P mode—a single `NOP` instructions must follow the `BFCLR` instruction, and the ‘>>’ assembler forcing operator must be used in the `JMP` instruction.

Code Example 5-17. Exiting Data-Memory Execution Mode, 19-Bit Target Address

```
BEGIN_X   EQU   $1000        ; Beginning address of program in data memory

          ORG   P:BEGIN_X,X:BEGIN_X     ; (code located in data memory)
          .
          .
          .
; Exact Sequence for Steps 1 through 3
          BFSET #$0300,SR    ; Disable Interrupts
          NOP                ; (wait for interrupts to be disabled)
          NOP                ; (wait for interrupts to be disabled)
          NOP                ; (wait for interrupts to be disabled)
          NOP                ; (wait for interrupts to be disabled)
          NOP                ; (wait for interrupts to be disabled)
          BFCLR #$0080,OMR   ; Disable data memory instruction fetches
          NOP                ; (wait for mode to switch)
          NOP                ; (wait for mode to switch)
; NOTE: Must Use Assembler Forcing Operator - Forces 19-bit Address
          JMP   >PMEM_TARGET       ; Jump to 1st instruction in program memory
          NOP                ; (fetched but not executed)
          NOP                ; (fetched but not executed)
          NOP                ; (fetched but not executed)

          ORG  P:; (indicates code located in program memory)
PMEM_TARGET
          ; Remember to RE-Enable Interrupts
```

NOTE:

The code that is used to exit data-memory execution mode must contain the exact number of `NOP` instructions that is shown in Code Example 5-17. There can be *no* jumps or branches to instructions within this sequence.

5.15 Unsupported DSP56800 Instruction Syntax

An undocumented and unsupported syntax for the `LEA` instruction is accepted by the DSP56800 assembler, but rejected by the DSP56800E assembler with an error message indicating that the DSP56800E only accepts supported DSP56800 syntax. In this syntax, an effective address is specified, followed by a comma and a destination register that is the same as the effective address. Examples of this unsupported syntax are listed in Code Example 5-18.

Code Example 5-18. Unsupported DSP56800 LEA Instruction Syntax

```
        LEA     (R2)+,R2     ; effective addr and dest are the same reg
        LEA     (R1)-,R1     ; effective addr and dest are the same reg
        LEA     (R0)+N,R0    ; effective addr and dest are the same reg
        LEA     (R2+7),R2    ; effective addr and dest are the same reg
        LEA     (R3+987),R3  ; effective addr and dest are the same reg
```

These same instructions are rewritten with the supported and correct DSP56800 instruction syntax in Code Example 5-19. The DSP56800E assembler accepts these corrected versions.

Code Example 5-19. Correct DSP56800 LEA Instruction Syntax

```
        LEA     (R2)+        ; no destination register specified
        LEA     (R1)-        ; no destination register specified
        LEA     (R0)+N       ; no destination register specified
        LEA     (R2+7)       ; no destination register specified
        LEA     (R3+987)     ; no destination register specified
```

5.16 Requirements on Context Save/Restore

NOTE:

This section only applies to code that mixes 56800 and 56800E instructions and makes use of the `MOVE.L` instruction to store or retrieve values to or from the stack.

Context switching can occur at the start of an interrupt service routine or when a function is called. In the DSP56800 architecture, if the SR or OMR register plus the hardware stack must be saved, the SR or OMR must be saved before the HWS is saved. This requirement is necessary because any value written to the hardware stack, either through a `DO` (or `DOSLC`) instruction or a `MOVE` instruction, causes the core to preserve the looping state, which means that the SR's LF bit is copied to the OMR's NL bit, and the LF bit is then set. This sequence is prescribed in the implementation of hardware loops for both architectures.

Additional resources are introduced in the DSP56800E architecture to support nested loop capability. When a value is written to the primary loop counter register, the core writes the original value to the second loop counter register, LC2. Therefore, LC2 must be saved before LC to guarantee proper restoration.

Typically, a full context switch is not required in most applications, but in the DSP56800E architecture the following rules must always be followed:

- For Full or Partial Context Save:
  - If the `MOVE.L` instruction is used to save registers in the stack, the stack pointer SP, must be odd-aligned. For example, if SP is $1001, the lower 16-bits of the pointed element resides in address $1000, and the upper 16-bits resides in address $1001.
  - If LC must be preserved, LC2 must be saved before LC is saved.
  - If the HWS must be preserved, the SR and OMR registers must be saved before the HWS is saved.
  - If the hardware LIFO stack must be preserved, the HWS must be saved twice in order to save HWS0 followed by HWS1.
  - When the complete accumulator is saved with `MOVE.L`, the following sequence must be followed:

```
MOVE.L  A2,X:(SP)+          ; Save 4-bit extension register A2
MOVE.L  A10,X:(SP)+         ; Save 32-bit A10
```

- In Context Restore:
  - When LC is preserved, LC2 must be restored after LC is restored.
  - If HWS is preserved, the SR and OMR registers must be restored after HWS is restored.
  - If the hardware LIFO stack is preserved, the HWS must be restored twice in order to restore HWS0 and HWS1.
  - When the complete accumulator is saved with MOVE.L, the following sequence must be followed:

```
MOVE.L  X:(SP)-,A           ;Load 32 bits A10 with sign extension
MOVE.L  X:(SP)-,A2          ;Load 4-bit extension register A2
```

Code Example 5-20 and Code Example 5-21 present code for full context save and full context restore respectively in the DPS56800E architecture.

Code Example 5-20. Full Context Save for DSP56800E

```
; Exact Sequence for Full Context Save

FContext_FullSave:
        ADDA    #2,SP        ; Point to Empty Location

        MOVE.L  N,X:(SP)+    ;
        MOVE.L  R0,X:(SP)+   ;
        MOVE.L  R1,X:(SP)+   ;
        MOVE.L  R2,X:(SP)+   ;
        MOVE.L  R3,X:(SP)+   ;
        MOVE.L  R4,X:(SP)+   ;
        MOVE.L  R5,X:(SP)+   ;
        MOVE.L  LA,X:(SP)+   ;
        MOVE.L  LA2,X:(SP)+  ;
        MOVE.L  LC2,X:(SP)+  ; LC2 must be saved before LC
        MOVE.L  LC,X:(SP)+   ;
        MOVE.L  A2,X:(SP)+   ;
        MOVE.L  OMR,X:(SP)+  ; OMR must be saved before HWS
        MOVE.L  A10,X:(SP)+  ;
        MOVE.L  B2,X:(SP)+   ;
        MOVE.L  SR,X:(SP)+   ; SR must be saved before HWS
        MOVE.L  B10,X:(SP)+  ;
        MOVE.L  C2,X:(SP)+   ;
        MOVE.L  M01,X:(SP)+  ;
        MOVE.L  C10,X:(SP)+  ;
        MOVE.L  D2,X:(SP)+   ;
        MOVE.L  N3,X:(SP)+   ;
        MOVE.L  D10,X:(SP)+  ;
        MOVE.L  HWS,X:(SP)+  ; HWS0 is saved and HWS1 is written to HWS0
        MOVE.L  HWS,X:(SP)+  ; HWS1 is saved
        MOVE.L  X0,X:(SP)+   ;
        MOVE.L  Y,X:(SP)     ;

        ; Body of called ISR (or subroutine)
```

Code Example 5-21. Full Context Restore for DSP56800E

```
; Exact Sequence for Full Context Restore

FContext_FullRestore:

        MOVE.L  X:(SP)-,Y    ;
        MOVE.L  X:(SP)-,X0   ;
        MOVE.L  X:(SP)-,HWS  ; Value written to HWS0
        MOVE.L  X:(SP)-,HWS  ; HWS0 written to HWS1, HWS0 restored
        MOVE.L  X:(SP)-,D    ;
        MOVE.L  X:(SP)-,N3   ;
        MOVE.L  X:(SP)-,D2   ;
        MOVE.L  X:(SP)-,C    ;
        MOVE.L  X:(SP)-,M01  ;
        MOVE.L  X:(SP)-,C2   ;
        MOVE.L  X:(SP)-,B    ;
        MOVE.L  X:(SP)-,SR   ; SR restored after restoring HWS
        MOVE.L  X:(SP)-,B2   ;
        MOVE.L  X:(SP)-,A    ;
        MOVE.L  X:(SP)-,OMR  ; OMR restored after restoring HWS
        MOVE.L  X:(SP)-,A2   ;
        MOVE.L  X:(SP)-,LC   ;
        MOVE.L  X:(SP)-,LC2  ;
        MOVE.L  X:(SP)-,LA2  ;
        MOVE.L  X:(SP)-,LA   ;
        MOVE.L  X:(SP)-,R5   ;
        MOVE.L  X:(SP)-,R4   ;
        MOVE.L  X:(SP)-,R3   ;
        MOVE.L  X:(SP)-,R2   ;
        MOVE.L  X:(SP)-,R1   ;
        MOVE.L  X:(SP)-,R0   ;
        MOVE.L  X:(SP)-,N    ; SP points to last location used
                             ; before context save

                             ; End Of FullRestore
```

5.17 Legacy Programs Larger than 64K

Legacy programs that grow beyond the 64K word boundary are not supported by the CodeWarrior assembler. In order to port these programs, the user must translate the application to the new instruction set and re-assemble the new file without the use of the legacy switch.

5.18 Compatibility Issues at the Chip Level

There are other issues that can also affect compatibility which appear at the chip level. These are related to the Interrupt Priority Levels, where the DSP56800E has 2 extra levels in addition to the LP level, the lowest priority level. Compatibility issues at the Chip Level will also appear in the Interrupt Vector Locations and Peripheral Space Locations which are controlled by the chip implementations. The DSP56800 restricted simultaneous accesses from the XAB2 nd XDB2 buses when such accesses were done to off-chip data memory. There is no such restriction in the new DSP56800E architecture.

5.18.1 Interrupt Priority Level

The number of interrupt priority levels has increased from two levels on the DSP56800 to four levels on the DSP56800E. To guarantee exact compatibility between the two architectures, there is an exact mapping between the DSP56800's priority levels to those on DSP56800E:

- DSP56800 Priority Level 0 maps to Level 1 on DSP56800E
- DSP56800 Priority Level 1 maps to Level 3 on DSP56800E

The second mapping from level 1 to level 3 is a non-maskable priority level on both architectures.

The interrupt mask bits I1 and I0 (SR bits 9 and 8) reflect the current priority level of the DSC core and indicate the interrupt priority level (IPL) required for an interrupt source to interrupt the processor. Table 5-1 illustrates the mapping between the DSP56800 and the DSP56800E:

Table 5-1. Mapping the DSP56800 Interrupt Levels to the DSP56800E

I1	I0	DSP56800 Interrupt Priority Levels	Corresponding DSP56800E Interrupt Priority Levels
0	0	(Reserved)	SWILP instr. (maskable)
			Level 0 (maskable)
0	1	Level 0 (maskable)	Level 1 (maskable)
1	0	(Reserved)	Level 2 (maskable)
1	1	Level 1 (nonmaskable)	Level 3 (nonmaskable)

NOTE:

In the DSP56800E, the lowest priority level, LP, can only be generated by the SWILP instruction. The highest priority, level 3 interrupts can only be generated by the core and are nonmaskable. When an exception or interrupt is recognized and the current core priority level (CCPL) is lower than the incoming level, the CCPL is automatically updated to be one higher than the level of the incoming interrupt (except for the case of SWILP, which does not update the CCPL, or the case of level 3 interrupts, which leave the priority level at level 3).

Table 5-2 shows which levels of interrupts are accepted and which are masked for the four different DSP56800E current core priority levels (CCPL).

Table 5-2. Interrupt Mask Bit Definition

I1	I0	CCPL	DSP56800		DSP56800E	
			Exceptions Permitted	Exceptions Masked	Exceptions Permitted	Exceptions Masked
0	0	0	(Reserved)	(Reserved)	IPL 0, 1, 2, 3 and SWILP	None
0	1	1	IPL 0, 1	None	IPL 1, 2, 3	IPL 0 and SWILP
1	0	2	(Reserved)	(Reserved)	IPL 2, 3	IPL 0, 1 and SWILP
1	1	3	IPL 1	IPL 0	IPL 3	IPL 0, 1, 2 and SWILP

5.18.2 Interrupt Vector Locations

The DSP56800E architecture does not restrict the location of interrupt vectors. If exact compatibility is required at the chip level, the interrupt vectors should be in the exact same locations as for DSP56800 chip implementations.

5.18.3 Peripheral Space Locations

The DSP56800E architecture does not restrict the locations of on-chip peripherals memory mapped into data memory. If exact compatibility is required at the chip level, these registers should be in the exact same locations as for DSP56800 chip implementations.

5.18.4 Dual Read Instruction

The DSP56800E core does not restrict the memory access performed on the XAB2 and XDB2 buses, i.e., the memory accessed performed as the second read in a dual read instruction. This is defined by the implementation of the particular DSP56800E core-based chip.

The DSP56800 architecture specifies that the second read in a Dual Parallel Read instruction is always done to on-chip memory, and can never access on-chip peripherals or off-chip data memory.

A potential compatibility issue can occur when an address outside the data memory range is accessed using the XAB2 / XDB2 buses. If compatibility in this area is required for a specific DSP56800E-based chip, that chip should be designed with the DSP56800 restrictions—no accesses are permitted to on-chip peripherals or off-chip data memory. This is not an issue for DSP56800 applications where the address of a second read of a Dual Read instruction always accesses on-chip data memory.

5.18.5 The OMR EX Bit

The DSP56800E core does not define the precise operation of the EX bit in the OMR. The exact behavior of the EX bit for a given device depends on the device implementation. Consult the appropriate device’s user’s manual for more information on the EX bit.

The DSP56800 architecture uses the EX bit to remap on-chip data memory to off-chip data memory except for locations accessed with the X:<<pp addressing mode.

A potential compatibility issue can occur if a DSP56800E chip does not configure the EX bit to remap data memory *and* the original DSP56800 application uses the EX bit. If compatibility in this area is required for a specific DSP56800E-based chip, that chip should be designed with DSP56800 behavior—data memory can be remapped to off-chip using the EX bit.

5.19 Delay on Interrupt Enable and Disable

5.19.1 Enabling Interrupts — CCPL set to “0”

It is always recommended in both architectures that `MOVE` instructions not be used to change the Status Register’s I1 and I0 bits. Instead, interrupts are typically enabled using the `BFCLR` instruction. In Section 5.18.1 on page 5–21, it was shown that IPL 0 (the lowest priority level in the DSP56800 architecture), is mapped to IPL 1 in the DSP56800E architecture. Interrupt enabling in the DSP56800 is performed by setting {I1,I0} in the SR register to {0,1}, (interrupt mask bit I0 was required to always be written with a “1” to ensure future compatibility with future family members). Note that this would correspond to IPL 1 in the DSP56800E architecture. As shown below, IPL 0 is programmed in the DSP56800E architecture by setting {I1,I0} in the SR register to {0,0}, i.e. CCPL to “0”.

Code Example 5-22. Enabling Interrupts for DSP56800E - Setting CCPL to “0”

```
        BFCLR   #$0300,SR    ; Clear I1, I0 bits in SR (bits 9,8) - To IPL 0
                             ; DSP56800E: all interrupt levels:
                             ; ---- IPL 0, 1, 2, 3 and SWILP are permitted
```

Code Example 5-23. Enabling Interrupts for DSP56800

```
        BFSET   #$0300,SR    ; Modify I1, I0 bits in SR (bits 9,8)
        BFCLR   #$0200,SR    ; I0 must be 1 (bit 8) - To IPL 0
                             ; DSP56800: all interrupt levels:
                             ; ---- IPL 0 and 1 are permitted
```

In the DSP56800E architecture, there is a delay which occurs between the execution of the `BFCLR` instruction and the point where the interrupt arbiter sees the CCPL with the value “0”. Instructions in the following six clock cycles (any hardware stall cycles are also counted) will be executed before any pending interrupts; after the new CCPL is recognized, the interrupt is serviced. If the 6th clock cycle occurs during a multi-cycle instruction, interrupts can only be serviced after the completion of this instruction. This is demonstrated in Code Example 5-24. In the DSP56800, the delay is typically one or two cycles.

Code Example 5-24. Demonstrating Delay after Enabling Interrupts

```
; SWI #1 interrupt will remain pending until 6 cycles after enabling

        BFSET   #$0300,SR    ; Interrupts initially disabled

        NOP                  ;
        NOP                  ;
        SWI     #1           ; Generate Interrupt Request at Level 1
                             ; -- not yet taken because interrupts disabled
```

```
                      ; -- SWI #1 interrupt remains pending
      NOP             ;
      NOP             ;
      NOP             ;

      ...             ; (other instructions)

      BFCLR  #$0300,SR  ; Enable Interrupts
      INC.W  A          ; -> 1st one cycle instruction
      INC.W  A          ; -> 2nd one cycle instruction
      INC.W  A          ; -> 3rd one cycle instruction
      INC.W  A          ; -> 4th one cycle instruction
      INC.W  A          ; -> 5th one cycle instruction
      INC.W  A          ; -> 6th instruction, or any number of cycles
                        ; SWI #1 Interrupt Taken Here
      ASL    B          ; -> 7th instruction (executed after
                        ; returning from interrupt handler)
```

5.19.2 Disabling Interrupts — CCPL set to “3”

Interrupts are typically enabled using the `BFSET` instruction as shown below. In both architectures, this can be achieved with the single instruction, `BFSET`. In the DSP56800E, setting CCPL to “3” masks all Level 0, 1, 2 and LP exceptions. Disabling interrupts can be achieved in both architectures by setting {I1,I0} in the SR register to {1,1}.

Code Example 5-25. Disabling Interrupts - Setting CCPL to “3”

```
      BFSET  #$0300,SR  ; Set I1, I0 bits in SR (bits 9,8)
                        ; DSP56800E: only IPL 3 (non-maskable) permitted
                        ; DSP56800: only IPL 1 (non-maskable) permitted
```

In the DSP56800E architecture, there is a delay which occurs between the execution of the `BFSET` instruction and the point where the interrupt arbiter masks incoming interrupts. Even though interrupts are disabled using `BFSET`, interrupts can still be taken after any instruction completing execution anytime during the next five clock cycles (any hardware stall cycles are also counted) following the `BFSET` instruction. Instructions beginning execution in the 6th clock cycle and beyond will form the beginning of the non-interruptible sequence. This is demonstrated in Code Example 5-26. In the DSP56800, the delay is typically one or two cycles.

Code Example 5-26. Demonstrating Delay after Disabling Interrupts

```
      BFSET  #$0300,SR  ; Disable interrupts

      NOP               ; -> 1st clock cycle - can still be interrupted
      NOP               ; -> 2nd clock cycle - can still be interrupted
      NOP               ; -> 3rd clock cycle - can still be interrupted
      NOP               ; -> 4th clock cycle - can still be interrupted
      NOP               ; -> 5th clock cycle - can still be interrupted

; ----- Instructions after this point form a non-interruptible sequence
      INC.W  A          ; -> 1st NON-INTERRUPTIBLE instruction
      INC.W  A          ; -> 2nd NON-INTERRUPTIBLE instruction
      INC.W  A          ; -> 3rd NON-INTERRUPTIBLE instruction

      ...               ; (other instructions)
```

Chapter 6
Optimizing Legacy Code

The two primary criteria for optimizing code are codes size and cycle count. In the DSP56800 architecture, each instruction cycle requires two clock cycles. In the DSP56800E, this is reduced by half—each instruction cycle takes only one clock cycle. In general, porting an assembly program from the DSP56800 to the DSP56800E reduces the cycle count of the program. Direct translation of an assembly program does not generate immediate code reduction.

Several features which differentiate the DSP56800E from the DSP56800 can be used to reduce code size, including

- High level abstraction in some new instructions, e.g. ASL16, ASR16 and arithmetic parallel instructions
- A full set of data types—byte, 16-bit words and 32-bit long words
- More registers in the register file for the AGU and DALU blocks
- A more extensive instruction set, with added flexibility for each data type
- 19 new AGU instructions that can operate directly on the AGU register file
- A larger set of arithmetic parallel instructions, with inverted input on `MAC`

The most measurable effect on codesize and cycle count can be generated from adapting a given algorithm to the new architecture. However, several simple optimizations can easily be implemented without having to change a program or algorithm. The following list of optimizations, though not comprehensive, offers several methods that are easy to implement.

1. Use the two new C and D accumulators to perform spilling. The `TFR` instruction can copy one accumulator to another with saturation when the SA bit in the OMR is set.
2. Take advantage of the larger set of AGU registers, R0–R5 and N. N can also be used as a pointer register.
   Example: `MOVE.W  X0,X:(N)+`
3. Save an accumulator with a single long move rather than two 16-bit writes.
   Example: `MOVE.L  A10,X:(R0)+` where `R0` is even aligned.

   All long moves must conform with alignment rules specified in the *DSP56800E Core Reference Manual*. These rules include the following:

   — The lower 16-bits of a long is always stored in an even location.

   — All pointers (except SP) must be even aligned when used to point to 32-bit locations.

   — When the stack is used to store long data types, SP must always contain the odd value of the 32-bit address, i.e., point to the upper 16-bits of the long word.
4. Use the `ASR16` instruction to cast integers to longs.

5. Use the `ASL16` instruction to cast longs to integers.
6. Do not use DALU instructions to perform AGU arithmetic. There are 19 additional instructions in the 56800E that greatly facilitate address computations and compares.
7. Use the instruction `TST.W <Acc>` to clear the carry (C) bit in the SR. This instruction clears the C bit with no pipeline dependencies.
8. The DSP56800E architecture contains several delay instructions with 2 or 3 delay slots. These slots can be replaced with useful instructions that perform branches, function returns and interrupt returns, including `BRAD`, `JMPD`, `RTSD`, `RTID`. For a comprehensive set of rules governing the use of delay slots, refer to the *DSP56800E Core Reference Manual*.
9. Use the shadow registers when fast interrupt processing is not used.
10. Take advantage of the expanded set of arithmetic parallel instructions available in the instruction set.
11. Hardware `DO` loops can be nested up to two deep, accelerating complex algorithms. This avoids having to save the LC and LA registers before entering the inner loop.

Appendix A
Translation Tables

This appendix contains a complete set of tables showing how each DSP56800 instruction and instruction alias is mapped to a counterpart in the DSP56800E instruction set, as well as tables of legacy instructions and aliases. These detailed tables include every addressing mode available on the DSP56800. The CodeWarrior IDE switch to accept legacy instructions must be set in order for the assembler to recognize DSP56800 syntax.

First, a description of the register field notation used in the tables is presented. This is followed by the instruction mapping tables, the summary of Legacy instructions and instruction aliases.

A.1 Register Field Notation

The register field notations for the DSP56800 and DSP56800E architectures are very similar. This section presents both notations in order to correctly interpret the mapping, legacy, and alias tables. The tables for the DSP56800E presented here are a subset of those defined in the *DSP56800E Core Reference Manual*.

In some cases, the notation used to specify an accumulator determines whether or not saturation is enabled when the accumulator is being used as a source in a move or parallel move instruction. This is explained more fully in the sections in the *DSP56800E Core Reference Manual* titled “Data Limiter” and “Accessing the Accumulator Registers.”

Several kinds of register sets are used in the instruction summary tables. These sets are categorized for clarity as follows:

- General purpose writes and reads
- AGU registers
- DALU registers
- Additional register sets for Move instructions

A.1.1 DSP56800 Register Field Notation

Table A-1 shows the register sets available in the DSP56800 for the most important move instructions. Sometimes a register field is broken into two different fields, one where the register is used as a source (src), and the other where it is used as a destination (dst). This is important because different notations are used to store an accumulator value depending on saturation. In addition, the register fields in Table A-2 are used in Move instructions as sources and destinations within the AGU.

Table A-1. DSP56800 Register Fields for General Purpose Move Instructions

Register Field	Registers in this Field	Comments
8-HHHH	A, B, A1, B1 Y1, Y0, X0 R0-R3, N	Two ways to access the accumulator registers. All Data ALU and AGU registers are used as sources (or destinations) in MOVE instructions.

Table A-2 shows the register set available for use as pointers in address register indirect addressing modes. The most common fields used in this table are Rn and RRR. This table also shows the notation used for AGU registers in AGU arithmetic operations.

Table A-2. DSP56800 Address Generation Unit (AGU) Registers

Register Field	Registers in this Field	Comments
Rk	R0-R3 SP	Five AGU Registers available as pointers for addressing and address calculations.
Rj	R0, R1, R2, R3	Four Pointer Registers available as pointers for addressing. SP is excluded from this group.
N	N	Offset Register available for “Indexed by Offset” addressing mode and post-update.
M01	M01	Modifier register. Specifies whether linear or modulo arithmetic when a new address is calculated on R0 and R1.

Table A-3 shows the register set available for use in Data ALU arithmetic operations. The most common field used in this table is FDD.

Table A-3. DSP56800 Data ALU Registers

Register Field	Registers in this Field	Comments
FDD	A, B Y1, Y0, X0	Five data ALU registers—two 36-bit accumulators and three 16-bit data registers accessible during data ALU operations Contains the contents of the F and DD register fields.
F1DD	A1, B1 Y1, Y0, X0	Two 16-bit MSP portions of the accumulators, three 16-bit data registers. Contains the contents of the F1 and DD register fields.
DD	Y1, Y0, X0	Three 16-bit data registers
F	A, B	Two 36-bit accumulators accessible during parallel move instructions and some data ALU operations.
F1	A1, B1	The 16-bit MSP portion of two accumulators accessible as source operands.
Fj	A2, A1, A0 B2, B1, B0	The 4-bit and 16-bit portions of the two accumulators
~F	B, A	Specifies that the source operand is one of the two 36-bit accumulators and is not the same as the destination accumulator. If the destination register is the A accumulator, the source ~F is the B accumulator. If the destination register is the B accumulator, the source ~F is the A accumulator.

Table A-4 shows additional register fields definitions available for move and other classes of instructions.

Table A-4. DSP56800 Additional Register Fields for Move Instructions

Register Field	Registers in this Field	Comments
8-DDDDD	A, A2, A1, A0 B, B2, B1, B0 Y1, Y0, X0 R0, R1, R2, R3 N, SP M01 LA, LC, HWS OMR, SR	This table contains all of the DSP56800 CPU registers. It contains the contents of the 8-HHHHH and 8-SSSS register fields. This is a subset of the complete set of register in the DSP56800E, designated as DDDDD.
8-HHHHH	A, A2, A1, A0 B, B2, B1, B0 Y1, Y0, X0	This set designates registers which are written with signed values when written with word values. This set is a subset of those registers designated as HHHHH is the DSP56800E. This table lists the CPU registers excluding the registers in the 8-SSSS field: R0-R3, N, M01, SP, LA, LC, OMR, SR, HWS. The registers in this field and 8-SSSS combine to make the 8-DDDDD register field.
8-SSSS	R0, R1, R2, R3 N, SP M01 LA, LC, HWS OMR, SR	This set designates registers which are written with unsigned values when written with word values. This set is a subset of those registers designated as SSSS in the DSP56800E. The registers in this field and 8-HHHHH combine to make the 8-DDDDD register field.

A.1.2 DSP56800E Register Field Notation

The tables in this sections present the notation used to specify legal DSP56800E registers. These tables show only the subset of the DSP56800E instructions necessary to identify legacy mappings. For the complete notation, refer to the *DSP56800E Core Reference Manual*.

Table A-5 shows the register fields available for the most important move instructions. In some cases the supported set of registers varies depending on whether they are the source or destination of an operation. Register fields used in conjunction with AGU move instructions are listed in Table A-6.

Table A-5. DSP56800E Register Fields for General-Purpose Writes and Reads

Register Field	Registers in This Field	Comments
HHH (destination)	A, B, C, D Y Y1, Y0, X0	Seven data ALU registers—four 16-bit MSP portions of the accumulators and three 16-bit data registers used as destination registers. Note the usage of A, B, C, and D. Writing word data to the 32-bit Y register clears the Y0 portion.

Table A-5. DSP56800E Register Fields for General-Purpose Writes and Reads (Continued)

Register Field	Registers in This Field	Comments
HHHH (source)	A1, B1, C1, D1 Y1, Y0, X0 R0–R5, N	Seven data ALU and seven AGU registers used as source registers. Note the usage of A1, B1, C1, and D1.
HHHH (destination)	A, B, C, D Y Y1, Y0, X0 R0–R5, N	Seven data ALU and seven AGU registers used as destination registers. Note the usage of A, B, C, and D. Writing word data to the 32-bit Y register clears the Y0 portion.

Table A-6 shows the register sets available for use as pointers in address-register-indirect addressing modes. The most commonly used fields in this table are Rn and RRR. This table also shows the notation used for AGU registers in AGU arithmetic operations.

Table A-6. DSP56800E Address Generation Unit (AGU) Registers

Register Field	Registers in This Field	Comments
Rn	R0–R5 N SP	Eight AGU registers available as pointers for addressing and address calculations
RRR	R0–R5 N	Seven AGU registers available as pointers for addressing and as sources and destinations for move instructions
Rj	R0, R1, R2, R3	Four pointer registers available as pointers for addressing
N3	N3	One offset register available only for post-update in the second access of dual parallel read instructions
M01	M01	Address modifier register
FIRA	FIRA	Fast interrupt return register

Table A-7 shows the register sets available for use in data ALU arithmetic operations. The most commonly used fields in this table are EEE and FFF.

Table A-7. DSP56800E Data ALU Registers

Register Field	Registers in This Field	Comments
FFF	A, B, C, D Y Y1, Y0, X0	Eight data ALU registers—four 36-bit accumulators, one 32-bit long register Y, and three 16-bit data registers accessible during data ALU operations.
FFF1	A1, B1, C1, D1 Y1, Y0, X0	Seven data ALU registers—four 16-bit MSP portions of the accumulators and three 16-bit data registers accessible during data ALU operations. This field is identical to the HHH (source) field. It is very similar to FFF, but indicates that the MSP portion of the accumulator is in use. Note the usage of A1, B1, C1, and D1.
EEE	A, B, C, D Y1, Y0, X0	Seven data ALU registers—four accumulators and three 16-bit data registers accessible during data ALU operations. This field is similar to FFF but is missing the 32-bit Y register. Used for instructions where Y is not a useful operand (use Y1 instead).

Table A-7. DSP56800E Data ALU Registers (Continued)

Register Field	Registers in This Field	Comments
fff	A, B, C, D, Y	Four 36-bit accumulators and one 32-bit long register accessible during data ALU operations.
FF	A, B, C, D	Four 36-bit accumulators accessible during data ALU operations.
DD	Y1, Y0, X0	Three 16-bit data registers.
FF	A, B	Two 36-bit accumulators accessible during parallel move instructions and some data ALU operations.
F1	A1, B1	The 16-bit MSP portion of two accumulators accessible as source operands in parallel move instructions.
~F	B, A	Specifies that the source operand is one of the two 36-bit accumulators, A or B, and is not the same as the destination accumulator. (This register notation is only used for Tcc in the DSP56800E architecture). If the destination register is the A accumulator, the source ~F is the B accumulator. If the destination register is the B accumulator, the source ~F is the A accumulator.

Table A-8 shows additional register fields that are available for Move instructions.

Table A-8. DSP56800E Additional Register Fields for Move Instructions

Register Field	Registers in This Field	Comments
DDDDD	A, A2, A1, A0 B, B2, B1, B0 C, C1 D, D1 Y Y1, Y0, X0 R0, R1, R2, R3 R4, R5, N, SP M01, N3 LA, LC, HWS OMR, SR	This table lists the CPU registers. It contains the contents of the HHHHH and SSSS register fields. Y is permitted only as a destination, not as a source. Writing word data to the 32-bit Y register clears the Y0 portion. Note that the C2, C0, D2, and D0 registers are not available within this field. See the dd register field for these registers
HHHHH	A, A2, A1, A0 B, B2, B1, B0 C, C1 D, D1 Y Y1, Y0, X0	This set designates registers that are written with signed values when written with word values. Y is permitted only as a destination, not as a source register. The registers in this field and SSSS combine to make the DDDDD register field.

Table A-8. DSP56800E Additional Register Fields for Move Instructions (Continued)

Register Field	Registers in This Field	Comments
SSSS	R0, R1, R2, R3 R4, R5, N, SP M01, N3 LA, LC, HWS OMR, SR	This set designates registers that are written with unsigned values when written with word values. The registers in this field and HHHHH combine to make the DDDDD register field.

A.1.3 Immediate Value Notation

Immediate values, including absolute and offset addresses, are presented in the instruction set summary using the notation shown in Table A-9. These fields are used in “bit manipulation” and “change of flow” instructions.

Table A-9. Immediate Value Notation

Immediate Value Field	Description
<MASK16>	16-bit mask value
<MASK8>	8-bit mask value
<OFFSET18>	18-bit signed PC-relative offset
<OFFSET7>	7-bit signed PC-relative offset
<ABS16>	16-bit absolute address
<ABS19>	19-bit absolute address

A.2 Instruction Mapping Tables

This section provides a complete mapping of the entire instruction set of the DSP56800 into DSP56800E instructions. The mapping is organized in the following five tables:

- Table A-10 on page A-8—All DSP56800 instructions excluding Tcc, parallel moves, and aliases.
- Table A-11 on page A-21—The Tcc Instruction
- Table A-12 on page A-21—Single parallel move instructions
- Table A-13 on page A-22—Dual parallel read instructions
- Table A-14 on page A-23—Instruction aliases

NOTE:

The mapping tables do not contain new DSP56800E instructions which were not included in the DSP56800, such as `MOVE.B` and `ASLA`. The tables contain only the subset of DSP56800E instructions required to map from the DSP56800 architecture. In order for the assembler to recognize the DSP56800 syntax, the CodeWarrior IDE switch to accept legacy instructions must be set.

In certain instructions the name of the source accumulator changes from A1 (or B1) in the DSP56800 to A (or B) in the DSP56800E. For example, the `AND` instructions changes as follows:

- DSP56800 — `AND A1,X0`
- DSP56800E — `AND.W A,X0`

The following is a complete list of instructions in which F1 is mapped to F without changing the behavior of the operation on the DSP56800E:

- `AND F1,DD`
- `EOR F1,DD`
- `OR F1,DD`
- `ADD F1,DD`
- `SUB F1,DD`
- `CMP F1,DD`

The right-hand column of the mapping tables provides information regarding aspects of the translation to the DSP56800E architecture. The phrase “code growth” in this column indicates that the mapped opcode requires one additional program word in the new architecture. This is true in all cases except the branch instructions BRSET and BRCLR, where user intervention is required if the target label falls beyond the signed 7-bit offset representation, in which case the mapping requires two words.

Table A-10. Instruction Mapping: DSP56800 to DSP56800E

DSP56800			DSP56800E			Comments
Oper'n	Operands		Oper'n	Operands		
	Source(s)	Destination		Source(s)	Destination	
ABS	F		ABS	FFF		
ADC	Y	F	ADC	Y	F	
ADD	DD	FDD	ADD	FFF	FFF	F1 as source register maps to F in conversion.
	F1	DD				
	~F Y	F				
	#xx	FDD	ADD.W	#<0-31>	EEE	xx: [0,31]
	#xxxx			#xxxx		
	X:aa or X:<aa			X:xxxx		Code growth
	X:xxxx or X:>xx			X:xxxx		
	X:(SP-xx)			X:(SP-xx)		xx: [1,64]
	FDD	X:aa or X:<aa		EEE	X:xxxx	"ADD FDD,X:aa" no longer exists. No growth in code size.
		X:xxxx or X:>xx				
		X:(SP-xx)			X:(SP-xx)	xx: [1,64]
AND	DD	FDD	AND.W	EEE	EEE	F1 as source register maps to F in conversion.
	F1	DD				
ANDC	(operands)		(see Table A-14 on page A-23)			Alias
ASL	F		ASL	fff		
	DD		LSL.W	EEE		**Note:** Not "ASL.W " as expected.
ASLL	Y0,Y0	FDD	ASLL.W	Y0,Y0	FFF	
	Y1,Y0			Y1,Y0		
	A1,Y0			A1,Y0		
	B1,Y1			B1,Y1		
	Y1,X0			Y1,X0		
	Y0,X0			Y0,X0		
ASR	FDD		ASR	FFF		
ASRAC	Y0,Y0	F	ASRAC	Y0,Y0	FF	
	Y1,Y0			Y1,Y0		
	A1,Y0			A1,Y0		
	B1,Y1			B1,Y1		
	Y1,X0			Y1,X0		
	Y0,X0			Y0,X0		

Table A-10. Instruction Mapping: DSP56800 to DSP56800E (Continued)

DSP56800			DSP56800E			Comments
Oper'n	Operands		Oper'n	Operands		
	Source(s)	Destination		Source(s)	Destination	
ASRR	Y0,Y0	FDD	ASRR.W	Y0,Y0	FFF	
	Y1,Y0			Y1,Y0		
	A1,Y0			A1,Y0		
	B1,Y1			B1,Y1		
	Y1,X0			Y1,X0		
	Y0,X0			Y0,X0		
Bcc	aa or <aa		Bcc	<OFFSET7>		Default option
				><OFFSET18>		Use this option if link error encountered. (Selected by user if an unresolved reference encountered due to code growth; use the ">" forcing operator)
BFCHG	#xxxx	X:aa or X:<aa	BFCHG	#<MASK16>	X:aa	Same ID
		X:pp or X:<<pp			X:<<pp	
		X:xxxx or X:>xx			X:xxxx	
		X:(R2+xx)			X:(Rn+xxxx)	xx: [0,63]: code growth
		X:(SP-xx)			X:(SP-xx)	xx: [1,64]
		8-DDDDD			DDDDD	
BFCLR	#xxxx	X:aa or X:<aa	BFCLR	#<MASK16>	X:aa	Same ID
		X:pp or X:<<pp			X:<<pp	
		X:xxxx or X:>xx			X:xxxx	
		X:(R2+xx)			X:(Rn+xxxx)	xx: [0,63]: code growth
		X:(SP-xx)			X:(SP-xx)	xx: [1,64]
		8-DDDDD			DDDDD	
BFSET	#xxxx	X:aa or X:<aa	BFSET	#<MASK16>	X:aa	Same ID
		X:pp or X:<<pp			X:<<pp	
		X:xxxx or X:>xx			X:xxxx	
		X:(R2+xx)			X:(Rn+xxxx)	xx: [0,63]: code growth
		X:(SP-xx)			X:(SP-xx)	xx: [1,64]
		8-DDDDD			DDDDD	

Table A-10. Instruction Mapping: DSP56800 to DSP56800E (Continued)

DSP56800			DSP56800E			Comments
Oper'n	Operands		Oper'n	Operands		
	Source(s)	Destination		Source(s)	Destination	
BFTSTH	#xxxx	X:aa or X:<aa	BFTSTH	#<MASK16>	X:aa	Same ID
		X:pp or X:<<pp			X:<<pp	
		X:xxxx or X:>xx			X:xxxx	
		X:(R2+xx)			X:(Rn+xxxx)	xx: [0,63]: code growth
		X:(SP-xx)			X:(SP-xx)	xx: [1,64]
		8-DDDDD			DDDDD	
BFTSTL	#xxxx	X:aa or X:<aa	BFTSTL	#<MASK16>	X:aa	Same ID
		X:pp or X:<<pp			X:<<pp	
		X:xxxx or X:>xx			X:xxxx	
		X:(R2+xx)			X:(Rn+xxxx)	xx: [0,63]: code growth
		X:(SP-xx)			X:(SP-xx)	xx: [1,64]
		8-DDDDD			DDDDD	
BRA	aa or <aa		BRA	<OFFSET7>		Default option
				><OFFSET18>		Use this option if an unresolved reference is generated due to code growth; use the ">" forcing operator.

Table A-10. Instruction Mapping: DSP56800 to DSP56800E (Continued)

DSP56800			DSP56800E			Comments
Oper'n	Operands		Oper'n	Operands		
	Source(s)	Destination		Source(s)	Destination	
BRCLR	#MASK8	X:aa,AA or X:<aa,AA	BRCLR	#<MASK8>	X:aa, <OFFSET7>	7-Bit Offset Destination
		X:pp,AA or X:<<pp,AA			X:<<pp, <OFFSET7>	7-Bit Offset Destination
		X:xxxx,AA or X:>xx,AA			X:xxxx, <OFFSET7>	7-Bit Offset Destination
		X:(R2+xx),AA			X:(Rn+xxxx), <OFFSET7>	xx: [0,63]: code growth 7-Bit Offset Destination
		X:(SP-xx),AA			X:(SP-xx), <OFFSET7>	xx: [1,64] 7-Bit Offset Destination
		8-DDDDD,AA			DDDDD, <OFFSET7>	7-Bit Offset Destination

Note: For all addressing modes of BRCLR, if the location of the target label AA requires more than a 7-bit offset, <OFFSET7>, the following instruction pair must replace "`BRCLR #MASK8,<source>,AA`" to prevent an unresolved reference. The user must make use of forcing operator ">" on target label AA (in `BCS`) to generate the larger 18-bit target offset, <OFFSET18>. Code growth of two words always results from this mapping.

```
BFTSTL       #<MASK16>,<source>  ; test masked bits and set carry bit
                                 ;   if all bits are set
BCS           ><OFFSET18>        ; branch on condition of carry bit set
```

DSP56800			DSP56800E			Comments
BRSET	#MASK8	X:aa,AA or X:<aa,AA	BRSET	#<MASK8>	X:aa, <OFFSET7>	7-Bit Offset Destination
		X:pp,AA or X:<<pp,AA			X:<<pp, <OFFSET7>	7-Bit Offset Destination
		X:xxxx,AA or X:>xx,AA			X:xxxx, <OFFSET7>	7-Bit Offset Destination
		X:(R2+xx),AA			X:(Rn+xxxx), <OFFSET7>	xx: [0,63]: code growth 7-Bit Offset Destination
		X:(SP-xx),AA			X:(SP-xx), <OFFSET7>	xx: [1,64] 7-Bit Offset Destination
		8-DDDDD,AA			DDDDD, <OFFSET7>	7-Bit Offset Destination

Note: For all addressing modes of BRSET, if the location of the target label AA requires more than a 7-bit offset, <OFFSET7>, the following instruction pair must replace "`BRSET #MASK8,<source>,AA`" to prevent an unresolved reference. The user must make use of forcing operator ">" on target label AA (in `BCS`) to generate the larger 18-bit target offset, <OFFSET18>. Code growth of two words always results from this mapping.

```
BFTSTH       #<MASK16>,<source>  ; test masked bits and set carry bit
                                 ;   if all bits are set
BCS           ><OFFSET18>        ; branch on condition of carry bit set
```

Table A-10. Instruction Mapping: DSP56800 to DSP56800E (Continued)

DSP56800			DSP56800E			Comments
Oper'n	Operands		Oper'n	Operands		
	Source(s)	Destination		Source(s)	Destination	
CLR	F		CLR	F		
	{DD,Rj,N}		CLR.W	DDDDD		Alias
CMP	~F DD	F	CMP	EEE	EEE	
	#xx			#<0-31>	FF	xx: [0,31]
	#xxxx			#xxxx		
	X:aa or X:<aa			X:xxxx		Code growth
	X:xxxx or X:>xx			X:xxxx		
	X:(SP-xx)			X:(SP-xx)		xx: [1,64]
	DD	DD	CMP	EEE	EEE	F1 as source register maps to F in conversion.
	F1					
	#xx		CMP.W	#<0-31>		xx: [0,31]
	#xxxx			#xxxx		
	X:aa or X:<aa			X:xxxx		Code growth
	X:xxxx or X:>xx			X:xxxx		
	X:(SP-xx)			X:(SP-xx)		xx: [1,64]
DEBUG			DEBUGEV			
DEC or DECW	FDD		DEC.W	EEE		
	X:aa or X:<aa			X:xxxx		Code growth
	X:xxxx or X:>xx			X:xxxx		
	X:(SP-xx)			X:(SP-xx)		xx: [1,64]
DIV	DD	F	DIV	FFF1	fff	
DO	#xx	xxxx	DO	#<1-63>	<ABS16>	xx: [1,63]
	FDD Fj Rj N			DDDDD		
	LC		DOSLC	<ABS16>		Body of DOSLC Loop requires 2 instruction words; if required, a NOP instruction will be added by assembler to comply.
	LA		(NOT SUPPORTED)			
ENDDO			ENDDO			

Table A-10. Instruction Mapping: DSP56800 to DSP56800E (Continued)

DSP56800			DSP56800E			Comments
Oper'n	Operands		Oper'n	Operands		
	Source(s)	Destination		Source(s)	Destination	
EOR	DD	FDD	EOR.W	EEE	EEE	F1 as source register maps to F in conversion.
	F1	DD				
EORC	(operands)		(see Table A-14 on page A-23			Alias
ILLEGAL			ILLEGAL			
IMPY or IMPY16	Y0,Y0	FDD	IMPY.W	Y0,Y0	FFF	
	Y1,Y0			Y1,Y0		
	A1,Y0			A1,Y0		
	B1,Y1			B1,Y1		
	Y1,X0			Y1,X0		
	Y0,X0			Y0,X0		
INC or INCW	FDD		INC.W	EEE		
	X:aa or X:<aa			X:xxxx		Code Growth
	X:xxxx or X:>xx			X:xxxx		
	X:(SP-xx)			X:(SP-xx)		xx: [1,64]
Jcc	xxxxx		Jcc	<ABS19>		
JMP	xxxxx		JMP	<ABS19>		
JSR	xxxxx		JSR	<ABS19>		

LEA	(Rk)+N		**Note:** This sequence may be required:
			`SXTA.W N (See NOTE 1)`
			`ADDA N,Rn`
						For Rk=SP only. Code growth

			LEA	(Rj)+N		For Rk= R0-R3 only.
	(Rk+xxxx)		ADDA	#xxxx	Rn	For Rk=SP only.
				#<0-15>,Rn	(if possible)	For Rk=SP only. xxxx: [0,15]
			LEA	(Rj+xxxx)		For Rk= R0-R3 only.
	(R2+xx)		LEA	(Rj+xxxx)		xx: [0,63]: code growth
	(Rk)+		ADDA	#<0-15>	Rn	AGU reg increment by 1.
	(Rk)-		DECA	Rn		AGU reg decrement by 1.
	(SP-xx)		SUBA	#<1-64>	SP	xx: [1,64]
LSL	FDD		LSL.W	EEE		
LSLL	(operands)		(see Table A-14 on page A-23			Alias
LSR	FDD		LSR.W	EEE		

Table A-10. Instruction Mapping: DSP56800 to DSP56800E (Continued)

DSP56800			DSP56800E			Comments
Oper'n	Operands		Oper'n	Operands		
	Source(s)	Destination		Source(s)	Destination	
LSRAC	Y0,Y0	F	LSRAC	Y0,Y0	FF	
	Y1,Y0			Y1,Y0		
	A1,Y0			A1,Y0		
	B1,Y1			B1,Y1		
	Y1,X0			Y1,X0		
	Y0,X0			Y0,X0		
LSRR	Y0,Y0	FDD	LSRR.W	Y0,Y0	FFF	
	Y1,Y0			Y1,Y0		
	A1,Y0			A1,Y0		
	B1,Y1			B1,Y1		
	Y1,X0			Y1,X0		
	Y0,X0			Y0,X0		
MAC	(±)Y0,Y0	FDD	MAC	(±)Y0,Y0	FFF	
	(±)Y1,Y0			(±)Y1,Y0		
	(±)A1,Y0			(±)A1,Y0		
	(±)B1,Y1			(±)B1,Y1		
	(±)Y1,X0			(±)Y1,X0		
	(±)Y0,X0			(±)Y0,X0		
MACR	(±)Y0,Y0	FDD	MACR	(±)Y0,Y0	FFF	
	(±)Y1,Y0			(±)Y1,Y0		
	(±)A1,Y0			(±)A1,Y0		
	(±)B1,Y1			(±)B1,Y1		
	(±)Y1,X0			(±)Y1,X0		
	(±)Y0,X0			(±)Y0,X0		
MACSU	Y0,Y0	FDD	MACSU	Y0,Y0	EEE	
	Y0,Y1			Y0,Y1		
	Y0,A1			Y0,A1		
	Y1,B1			Y1,B1		
	X0,Y1			X0,Y1		
	X0,Y0			X0,Y0		

Table A-10. Instruction Mapping: DSP56800 to DSP56800E (Continued)

DSP56800			DSP56800E			Comments
Oper'n	Operands		Oper'n	Operands		
	Source(s)	Destination		Source(s)	Destination	
MOVE or MOVEC	X:(Rk+xxxx)	8-HHHHH	MOVE.W	X:(Rn+xxxx)	HHHHH	For Rk=SP only.
				X:(SP-xx)HHH	(if possible)	For Rk=SP only,xx: [1,64]
		8-SSSS	MOVEU.W	X:(Rn+xxxx)	SSSS	For Rk=SP only.
				X:(SP-xx),RRR	(if possible)	For Rk=SP only,xx: [1,64]
		8-DDDDD	MOVE	X:(Rj+xxxx)	DDDDD	For Rk= R0-R3 only.

	X:(Rk+N)	8-HHHHH	**Note:** This sequence may be required:
			`SXTA.W N (See NOTE 1)`
			`MOVE.W X:(Rn+N),HHHHH`
						For Rk=SP only.
						Code growth

		8-SSSS	**Note:** This sequence may be required:
			`SXTA.W N (See NOTE 1)`
			`MOVEU.W X:(Rn+N),SSSS`
						For Rk=SP only.
						Code growth

		8-DDDDD	MOVE	X:(Rj+N)	DDDDD	For Rk= R0-R3 only.
	X:(R2+xx)	8-HHHH	MOVE	X:(Rj+xxxx)	DDDDD	xx: [0,63]: code growth
	X:(Rk)	8-HHHHH	MOVE.W	X:(Rn)	HHHHH	
	X:(Rk)-			X:(Rn)-		
	X:(Rk)+			X:(Rn)+		
	X:(Rk)+N			X:(Rn)+N		
	X:xxxx or X:>xx			X:xxxx		

Table A-10. Instruction Mapping: DSP56800 to DSP56800E (Continued)

DSP56800			DSP56800E			Comments
Oper'n	Operands		Oper'n	Operands		
	Source(s)	Destination		Source(s)	Destination	
MOVE or MOVEC	X:(Rk)	8-SSSS	MOVEU.W	X:(Rn)	SSSS	
	X:(Rk)-			X:(Rn)-		
	X:(Rk)+			X:(Rn)+		
	X:(Rk)+N			X:(Rn)+N		
	X:xxxx or X:>xx			X:xxxx		
	X:(SP-xx)	FDD	MOVE.W	X:(SP-xx)	HHH	xx: [1,64]
		F1		X:(Rn+xxxx)	DDDDD	xx: [1,64]: code growth
		Rj N	MOVEU.W	X:(SP-xx)	RRR	xx: [1,64]
	8-DDDDD	X:(Rk+xxxx)	MOVE.W	DDDDD	X:(Rn+xxxx)	For Rk=SP only.
				HHHH,X:(SP-xx) (if possible)		For Rk=SP only,xx: [1,64]
			MOVE	DDDDD	X:(Rj+xxxx)	For Rk=R0-R3 only.

	8-DDDDD	X:(Rk+N)	**Note:** This sequence may be required:
			`SXTA.W N (See NOTE 1)`
			`MOVE.W DDDDD,X:(Rn+N)`
						For Rk=SP only.
						Code growth

			MOVE	DDDDD	X:(Rj+N)	For Rk=R0-R3 only.
	8-HHHH	X:(R2+xx)	MOVE	DDDDD	X:(Rj+xxxx)	xx: [0,63]: code growth
	8-DDDDD	X:(Rk)	MOVE.W	DDDDD	X:(Rn)	
		X:(Rk)-			X:(Rn)-	
		X:(Rk)+			X:(Rn)+	
		X:(Rk)+N			X:(Rn)+N	
		X:xxxx or X:.>xx			X:xxxx	
MOVE or MOVEC	F	X:(SP-xx)	MOVE.W	DDDDD	X:(Rn+xxxx)	xx: [1,64]: code growth
	F1DD Rj N			HHHH (source)	X:(SP-xx)	xx: [1,64]
	8-DDDDD	8-HHHHH	MOVE.W	DDDDD	HHHHH	Register to register move
		8-SSSS	MOVEU.W	DDDDD	SSSS	Can not use MOVE.W instruction here, to avoid sign extension and incompatibility.

Table A-10. Instruction Mapping: DSP56800 to DSP56800E (Continued)

DSP56800			DSP56800E			Comments
Oper’n	Operands		Oper’n	Operands		
	Source(s)	Destination		Source(s)	Destination	
MOVE or MOVEI	#xx	N	MOVE.W	#<-64-63>	HHHH	xx: [-64,63]
		Rj		#<0-63>	HHHH	xx: [0,63]
			MOVEU.W	#xxxx	SSSS	xx: [-64,-1]: code growth
		F1	MOVE.W	#xxxx	HHHHH	xx: [-64,63]: code growth
		FDD		#<-64-63>	HHHH	xx: [-64,63]
	#xxxx or #>xx	8-SSSS	MOVEU.W	#xxxx	SSSS	
		8-HHHHH	MOVE.W		HHHHH	
		X:xxxx or X:>xx		#xxxx	X:xxxx	If constant within [-64,63] range, use optimal selection.
				#<-64-63>,X:xxxx (if possible)		
		X:(R2+xx)		#xxxx	X:(Rn+xxxx)	xx: [0,63]: code growth
		X:(SP-xx)			X:(SP-xx)	xx: [1,64]
MOVE or MOVEM	P:(Rj)+	{Rj,N}	MOVEU.W	P:(Rj)+	RRR	
	P:(Rj)+N			P:(Rj)+N		
	P:(Rj)+	{DD,F,F1}	MOVE.W	P:(Rj)+	{Y1,Y0, X0, A,B,C,A1,B1}	
	P:(Rj)+N			P:(Rj)+N		
	8-HHHH	P:(Rj)+		{Y1,Y0, X0, A,B,C,A1,B1 R0-R5,N}	P:(Rj)+	
		P:(Rj)+N			P:(Rj)+N	
MOVE or MOVEP	8-HHHH	X:pp or X:<<pp	MOVE.W	{Y1,Y0, X0, A,B,C,A1,B1 R0-R5,N}	X:<<pp	
	#xxxx			#xxxx		
	X:pp or X:<<pp	{DD,F,F1}		X:<<pp	{Y1,Y0, X0, A,B,C,A1,B1}	
		{Rj,N}	MOVEU.W		RRR	
MOVE or MOVES	8-HHHH	X:aa or X:<aa	MOVE.W	{Y1,Y0, X0, A,B,C,A1,B1 R0-R5,N}	X:aa	
	#xxxx			#xxxx		
	X:aa or X:<aa	{DD,F,F1}		X:aa	{Y1,Y0, X0, A,B,C,A1,B1}	
		{Rj,N}	MOVEU.W		RRR	

Table A-10. Instruction Mapping: DSP56800 to DSP56800E (Continued)

DSP56800			DSP56800E			Comments
Oper'n	Operands		Oper'n	Operands		
	Source(s)	Destination		Source(s)	Destination	
MPY	(±)Y0,Y0	FDD	MPY	(±)Y0,Y0	FFF	
	(±)Y1,Y0			(±)Y1,Y0		
	(±)A1,Y0			(±)A1,Y0		
	(±)B1,Y1			(±)B1,Y1		
	(±)Y1,X0			(±)Y1,X0		
	(±)Y0,X0			(±)Y0,X0		
MPYR	(±)Y0,Y0	FDD	MPYR	(±)Y0,Y0	FFF	
	(±)Y1,Y0			(±)Y1,Y0		
	(±)A1,Y0			(±)A1,Y0		
	(±)B1,Y1			(±)B1,Y1		
	(±)Y1,X0			(±)Y1,X0		
	(±)Y0,X0			(±)Y0,X0		
MPYSU	Y0,Y0	FDD	MPYSU	Y0,Y0	EEE	
	Y0,Y1			Y0,Y1		
	Y0,A1			Y0,A1		
	Y1,B1			Y1,B1		
	X0,Y1			X0,Y1		
	X0,Y0			X0,Y0		
NEG	F		NEG	FFF		
NOP			NOP			
NORM	R0	F	NORM	R0	F	
NOT	FDD		NOT.W	EEE		
NOTC	(operands)		(see Table A-14 on page A-23)			Alias
OR	DD	FDD	OR.W	EEE	EEE	F1 as source register maps to F in conversion.
	F1	DD				
ORC	(operands)		(see Table A-14 on page A-23)			Alias
POP	(operands)		(see Table A-14 on page A-23			Alias

Table A-10. Instruction Mapping: DSP56800 to DSP56800E (Continued)

DSP56800			DSP56800E			Comments
Oper'n	Operands		Oper'n	Operands		
	Source(s)	Destination		Source(s)	Destination	
REP	#xx		REP	#<0-63>		xx: [0,63]
	FDD Fj Rj N			DDDDD		
	LC		DOSLC	<ABS16>		Code growth. Body of DOSLC Loop requires 2 instruction words; a NOP instruction will be added by assem-bler to comply.
	LA		(NOT SUPPORTED)			
RND	F		RND	fff		
ROL	FDD		ROL.W	EEE		
ROR	FDD		ROR.W	EEE		
RTI			RTI			
RTS			RTS			
SBC	Y	F	SBC	Y	F	
STOP			STOP			
SUB	DD	FDD	SUB	FFF	FFF	F1 as source register maps to F in conversion.
	F1	DD				
	~F Y	F				
	#xx	FDD	SUB.W	#<0-31>	EEE	xx: [0,31]
	#xxxx			#xxxx		
	X:aa or X:<aa			X:xxxx		Code growth
	X:xxxx or X:>xx			X:xxxx		
	X:(SP-xx)			X:(SP-xx)		xx: [1,64]
SWI			SWI			
Tcc	(operands)		Tcc	(See Table A-11 on page A-21		
TFR	DD ~F	F	TFR	FFF	fff	
TST	F		TST	FF		

Table A-10. Instruction Mapping: DSP56800 to DSP56800E (Continued)

DSP56800			DSP56800E			Comments
Oper'n	Operands		Oper'n	Operands		
	Source(s)	Destination		Source(s)	Destination	
TSTW	X:(Rk+xxxx)		TST.W	X:(Rn+xxxx)		For Rk=SP only.
				X:(SP-xx) (used if possible)		For Rk=SP only,xx: [1,64]
			TSTW	X:(Rj+xxxx)		For Rk= R0-R3 only.

	X:(Rk+N)		**Note:** This sequence may be required:
			`SXTA.W N (See NOTE 1)`
			`TST.W X:(Rn+N)`
						For Rk=SP only.
						Code growth

			TSTW	X:(Rj+N)		For Rk= R0-R3 only.
	X:(R2+xx)		TSTW	X:(Rj+xxxx)		xx: [0,63]: code growth
	X:aa or X:<aa		TST.W	X:aa		Same ID
	X:pp or X:<<pp			X:<<pp		
	X:xxxx or X:>xx			X:xxxx		
	X:(Rk)			X:(Rn)		
	X:(Rk)-			X:(Rn)-		
	X:(Rk)+		TST.W	X:(Rn)+		
	X:(Rk)+N			X:(Rn)+N		
	X:(SP-xx)			X:(SP-xx)		xx: [1,64]
	8-DDDDD (except F,HWS)			DDDDD		
	F		TSTW	F		Sets L bit; CC based on saturated value of F.
	(Rk)-		TSTDECA.W	Rn		
WAIT			WAIT			

1. When N is used (or loaded) as an offset register, the value and sign information must be preserved as a 24-bit signed value to guarantee compatibility. There are a few legacy cases where N is required to be a 24-bit sign-extended offset register. Sign extension is correctly generated by immediately preceding the instructions listed below with `SXTA.W N`, which extends the sign of the N register to 24-bits. This instruction (one extra program word) is only required when N is loaded as an offset register in the original legacy code. Conversely, if the N register is loaded temporarily with an AGU pointer, i.e. not representing an offset register, sign extension is not recommended. In this case, the upper 8 bits must be zeroes; otherwise, it may generate the wrong effective address when used. The following DSP56800 instructions depicted below, uses N as the offset register and SP as the pointer to the stack. These legacy instructions are mapped to the DSP56800E instructions listed below. In all of these cases, the N register is always expected to be a true 24-bit signed value, (representing offset):

• `LEA`	`(SP)+N`	mapped to	`ADDA`	`N,SP`
• `MOVE`	`X:(SP+N),8-HHHHH`	mapped to	`MOVE.W`	`X:(SP+N),HHHHH`
• `MOVE`	`X:(SP+N),8-SSSS`	mapped to	`MOVEU.W`	`X:(SP+N),SSSS`
• `MOVE`	`8-DDDDD,X:(SP+N)`	mapped to	`MOVE.W`	`DDDDD,X:(SP+N)`
• `TSTW`	`X:(SP+N)`	mapped to	`TST.W`	`X:(SP+N)`

Table A-11. Tcc Instruction Mapping: DSP56800 to DSP56800E

DSP56800			DSP56800E			Cmts
Oper'n	Operands		Oper'n	Operands		
	1st Move	2nd Move		1st Move	2nd Move	
Tcc	~F,F		Tcc	~F,F		no AGU register move
		R0,R1			R0,R1	
	DD,F			DD,F		no AGU register move
		R0,R1			R0,R1	

Table A-12. Single Parallel Move Mapping: DSP56800 to DSP56800E

DSP56800				DSP56800E			
DALU Operation		Parallel Move		DALU Operation		Parallel Move	
Operation	Opcodes	Source	Destination	Operation	Opcodes	Source	Destination
MAC	Y1,X0,F Y0,X0,F Y1,Y0,F Y0,Y0,F A1,Y0,F B1,Y1,F	X: (Rj)+ X:(Rj)+N	A B A1 B1 X0 Y0 Y1	MAC	Y1,X0,F Y0,X0,F Y1,Y0,F Y0,Y0,F A1,Y0,F B1,Y1,F	X: (Rj)+ X:(Rj)+N	A B A1 B1 X0 Y0 Y1
MPY				MPY			
MACR				MACR			
MPYR				MPYR			
ADD	X0,F Y1,F Y0,F A,B B,A			ADD	X0,F Y1,F Y0,F A,B B,A		
SUB				SUB			
CMP				CMP			
TFR				TFR			
ABS	F			ABS	F		
ASL				ASL			
ASR				ASR			
CLR				CLR			
RND				RND			
TST				TST			
DEC or DECW				DEC.W			
INC or INCW				INC.W			
NEG				NEG			

Table A-12. Single Parallel Move Mapping: DSP56800 to DSP56800E (Continued)

DSP56800				DSP56800E			
DALU Operation		Parallel Move		DALU Operation		Parallel Move	
Operation	Opcodes	Source	Destination	Operation	Opcodes	Source	Destination
MAC MPY MACR MPYR	Y1,X0,F Y0,X0,F Y1,Y0,F Y0,Y0,F A1,Y0,F B1,Y1,F	A B A1 B1 X0 Y0 Y1	X: (Rj)+ X:(Rj)+N	MAC MPY MACR MPYR	Y1,X0,F Y0,X0,F Y1,Y0,F Y0,Y0,F A1,Y0,F B1,Y1,F	A B A1 B1 X0 Y0 Y1	X:(Rj)+ X:(Rj)+N
ADD SUB CMP TFR	X0,F Y1,F Y0,F A,B B,A			ADD SUB CMP TFR	X0,F Y1,F Y0,F A,B B,A		
ABS ASL ASR CLR RND TST DEC or DECW INC or INCW NEG	F			ABS ASL ASR CLR RND TST DEC.W INC.W NEG	F		

Table A-13. Dual Parallel Read Mapping: DSP56800 to DSP56800E

DSP56800						DSP56800E					
DALU Operation		1st Pll Read		2nd Pll Read		DALU operation		1st Pll Read		2nd Pll Read	
Oper'n	Op'nds	Src 1	Dst	Src 2	Dst	Oper'n	Op'nds	Src 1	Dst	Src 2	Dst
MAC MPY MACR MPYR	X0,Y1,F X0,Y0,F Y1,Y0,F	X:(R0)+ X:(R0)+N X:(R1)+ X:(R1)+N	Y0 Y1	X:(R3)+ X:(R3)-	X0	MAC MPY MACR MPYR	X0,Y1,F X0,Y0,F Y1,Y0,F	X:(R0)+ X:(R0)+N X:(R1)+ X:(R1)+N	Y0 Y1	X:(R3)+ X:(R3)-	X0
ADD SUB	X0,F Y1,F Y0,F					ADD SUB	X0,F Y1,F Y0,F				
MOVE						MOVE.W					

Table A-14. Instruction Alias Mapping: DSP56800 to DSP56800E

DSP56800			DSP56800E			Comments
Operation	Operands		Operation	Operands		
	Op1	Op2		Op1	Op2	
ANDC	#xxxx	X:aa or X:<aa	ANDC	#<MASK16>	X:aa	ANDC's alias uses BFCLR
		X:pp or X:<<pp			X:<<pp	
		X:xxxx or X:>xx			X:xxxx	
		X:(R2+xx)			X:(Rn+xxxx)	xx: [0,63]: code growth
		X:(SP-xx)			X:(SP-xx)	xx: [1,64]
		8-DDDDD			DDDDD	
EORC	#xxxx	X:aa or X:<aa	EORC	#<MASK16>	X:aa	EORC's alias uses BFCHG
		X:pp or X:<<pp			X:<<pp	
		X:xxxx or X:>xx			X:xxxx	
		X:(R2+xx)			X:(Rn+xxxx)	xx: [0,63]: code growth
		X:(SP-xx)			X:(SP-xx)	xx: [1,64]
		8-DDDDD			DDDDD	
LSLL	Y0,Y0	DD	ASLL.W	Y0,Y0	FFF	alias uses ASLL in DSP56800. LSLL is legacy only
	Y1,Y0			Y1,Y0		
	A1,Y0			A1,Y0		
	B1,Y1			B1,Y1		
	Y1,X0			Y1,X0		
	Y0,X0			Y0,X0		
NOTC	X:aa or X:<aa		NOTC	X:aa		NOTC's alias uses BFCHG
	X:pp or X:<<pp			X:<<pp		
	X:xxxx or X:>xx			X:xxxx		
	X:(R2+xx)			X:(Rn+xxxx)		xx: [0,63]: code growth
	X:(SP-xx)			X:(SP-xx)		xx: [1,64]
	8-DDDDD			DDDDD		
ORC	#xxxx	X:aa or X:<aa	ORC	#<MASK16>	X:aa	ORC's alias uses BFSET
		X:pp or X:<<pp			X:<<pp	
		X:xxxx or X:>xx			X:xxxx	
		X:(R2+xx)			X:(Rn+xxxx)	xx: [0,63]: code growth
		X:(SP-xx)			X:(SP-xx)	xx: [1,64]
		8-DDDDD			DDDDD	
POP			DECA	SP		Not a supported alias on 56800E
	8-DDDDD		MOVE.W	X:(SP)-	HHHHH	
			MOVEU.W	X:(SP)-	SSSS	

A.3 Legacy Instruction Summary Tables

Tables A-15 through A-17 list the instruction summary tables for the legacy instructions `LEA`, `MOVE`, and `TSTW`. The information in this section supplements the information found in the *DSP56800E Core Reference Manual*.

As explained in Section 5.1 on page 5–1, these instructions are intended to access memory locations only within the first 64K of data memory. For instructions that generate an address, the upper 8 bits of the address are forced to all zeros. The move of immediate data also follows the rule of placing zero extended 16-bit values into the AGU registers.

Note that each move instruction uses the `MOVE` mnemonic. This differentiates the instructions from their `MOVE.W` counterparts, which operate differently. Similarly, the `TSTW` mnemonic is used to differentiate these instructions from `TST.W`, the standard counterpart in the DSP56800E instruction set.

NOTE:

These instructions are intended only for compatibility with existing DSP56800 code. They are only allowed when the legacy switch is set. They are not allowed for new applications that use only DSP56800E syntax.

Table A-15. Move Word Instructions — Legacy Code

Operation	Source	Dest	C	W	Comments
MOVE	X:(Rj+N)	DDDDD	2	1	Forces 16-bit AGU arithmetic. Only for R0,R1,R2,R3. The SP register is not allowed.
	X:(Rj+xxxx)	DDDDD	2	2	
	DDDDD	X:(Rj+N)	2	1	
	DDDDD	X:(Rj+xxxx)	2	2	

Table A-16. Data ALU Arithmetic Instructions — Legacy Code

Operation	Operands	C	W	Comments
TSTW	F	1	1	Test 16-bit word in register. Limiting occurs when reading the accumulator before the test operation if the extension register is in use.
	X:(Rj+N)	2	1	Forces 16-bit AGU arithmetic. Only for R0,R1,R2,R3. The SP register is not allowed.
	X:(Rj+xxxx)	2	2	

Table A-17. AGU Arithmetic Instructions — Legacy Code

Operation	Operands	C	W	Comments
LEA	(Rj)+N	1	1	Add Rj to N, zero extend the upper 8 bits of the result, and place the result in Rj. Forces 16-bit AGU arithmetic. Only for R0,R1,R2 and R3. SP register is not allowed.
	(Rj+xxxx)	2	2	Add Rj to xxxx, zero extend the upper 8 bits of the result, and place the result in Rj. Forces 16-bit AGU arithmetic. Only for R0,R1,R2,R3. The SP register is not allowed.

A.4 DSP56800 Instruction Aliases

The DSP56800 assembly language contains several instruction aliases. An instruction alias is an instruction that is accepted by the assembler, but is actually assembled as another DSP56800 instruction and is also disassembled as this second instruction. These aliases are listed in Table A-14 on page A-23. Each of these instructions is also mapped in Table A-10 on page A-8.

The instruction aliases `ANDC`, `EORC`, `NOTC`, and `ORC` in the DSP56800 map to identical instructions in the DSP56800E. The remaining DSP56800 instruction aliases—`LSLL`, `ASL`, `CLR`, and `POP`—map to different instructions in the DSP56800E. These differences are summarized in Table A-18.

Table A-18. Summary of DSP56800 Instruction Aliases and Mapping

DSP56800 Instruction	Operands	Actual DSP56800 Instruction	Operands	Actual DSP56800E Instruction	Operands
LSLL	Y1,X0,DD Y0,X0,DD Y1,Y0,DD Y0,Y0,DD A1,Y0,DD B1,Y1,DD	ASLL	Y1,X0,FFF Y0,X0,FFF Y1,Y0,FFF Y0,Y0,FFF A1,Y0,FFF B1,Y1,FFF	ASLL.W	Y1,X0,DD Y0,X0,DD Y1,Y0,DD Y0,Y0,DD A1,Y0,DD B1,Y1,DD
ASL	DD	LSL	DD	LSL.W	DD
CLR	X0, Y1, Y0, A1, B1, R0-R3, N	MOVE #0,	X0, Y1, Y0, A1, B1, R0-R3, N	CLR.W	DDDDD
POP	DDDDD	MOVE	X:(SP)-,DDDDD	MOVE.W	X:(SP)-,DDDDD
	(none)	LEA	(SP)-	DECA	SP

NOTE:

The `LSLL`, `CLR`, and `POP` instructions are not recommended for new applications. Instead, use the corresponding DSP56800E instructions listed in Table A-18.

A.4.1 LSLL Instruction Alias

The `LSLL` instruction operates identically to an arithmetic left shift, so this instruction is assembled as an `ASLL` instruction on the DSP56800. When the DSP56800E assembler encounters the `LSLL` instruction in DSP56800 legacy code, it is assembled and disassembled using the DSP56800E's `ASLL.W` instruction.

NOTE:

This instruction alias is not recommended for use in new applications. Instead, use the DSP56800E instruction, `ASLL.W`.

A.4.2 ASL Instruction Alias

The ASL instruction operation is similar to a logical left shift for the Y1, Y0, and X0 registers, so this instruction is assembled as an LSL on the DSP56800. `ASL` and `LSL` perform identical shifting but set the condition codes differently. When the DSP56800E assembler encounters the `ASL` instruction in DSP56800 legacy code, it is assembled and disassembled using the DSP56800E's `LSL.W` instruction.

The ASL instruction is not remapped when the operand is one of the accumulator registers or the Y register.

NOTE:

The DSP56800 `ASL DD` instruction maps to `LSL.W` on the DSP56800E rather than `ASL.W DD` as expected. Two differences characterize the operation of these two instructions: the way condition codes are set, and the effect of the result when the saturation bit (SA) is set in OMR register. The `LSL.W` instruction is affected by the SA bit, but `LSL.W` is not affected. If saturation is desired in a new application using "arithmetic shift left," the `ASL.W` instruction must be used.

A.4.3 CLR Instruction Alias

The CLR instruction operates identically to an immediate move instruction using the value $0, so this instruction is assembled as a move instruction on the DSP56800 architecture. When encountering the CLR <register> instruction in DSP56800 legacy code, it is assembled and disassembled using the DSP56800E's `CLR.W` instruction.

NOTE:

This case does not apply to the `CLR` instruction when performed on the A or B accumulator. In the DSP56800 architecture, the condition codes are affected if the destination of the `CLR` instruction is one of the two 36-bit accumulators (A or B).

This instruction alias is not recommended for use in new applications. Instead, use the appropriate DSP56800E instruction, `CLR.W`. The condition codes are not affected by this instruction.

A.4.4 POP Instruction Alias

The `POP` instruction operates identically to a move from the stack with post-decrement. It is assembled as a `MOVE` instruction on the DSP56800 if an operand is specified, and as a `LEA` instruction if no operand is specified. When the DSP56800E assembler encounters the `POP` instruction in DSP56800 legacy code, it is assembled and disassembled as

• `MOVE.W`	`X:(SP)-,<register>`	operand specified
• `DECA`	`SP`	no operand specified.

NOTE:

This instruction alias is not recommended for use in new applications. Instead, use the appropriate DSP56800E instructions, `MOVE.W` or `DECA`.

Index

How to Reach Us:

Home Page:
www.freescale.com

E-mail:
support@freescale.com

USA/Europe or Locations Not Listed:
Freescale Semiconductor
Technical Information Center, CH370
1300 N. Alma School Road
Chandler, Arizona 85224
+1-800-521-6274 or +1-480-768-2130
support@freescale.com

Europe, Middle East, and Africa:
Freescale Halbleiter Deutschland GmbH
Technical Information Center
Schatzbogen 7
81829 Muenchen, Germany
+44 1296 380 456 (English)
+46 8 52200080 (English)
+49 89 92103 559 (German)
+33 1 69 35 48 48 (French)
support@freescale.com

Japan:
Freescale Semiconductor Japan Ltd.
Headquarters
ARCO Tower 15F
1-8-1, Shimo-Meguro, Meguro-ku,
Tokyo 153-0064
Japan
0120 191014 or +81 3 5437 9125
support.japan@freescale.com

Asia/Pacific:
Freescale Semiconductor Hong Kong Ltd.
Technical Information Center
2 Dai King Street
Tai Po Industrial Estate
Tai Po, N.T., Hong Kong
+800 2666 8080
support.asia@freescale.com

For Literature Requests Only:
Freescale Semiconductor Literature Distribution Center
P.O. Box 5405
Denver, Colorado 80217
1-800-441-2447 or 303-675-2140
Fax: 303-675-2150
LDCForFreescaleSemiconductor@hibbertgroup.com

DSP56800ERG
Rev. 1.1, 11/2005

freescale™
semiconductor