AN2715
Rev. 1, 11/2004

Porting Code From the DSP56300 Family of Products to the SC140/SC1400 Core

by Iantha Scheiwe

As evolving DSP56300-based applications require additional processing power, the natural evolution path leads to SC140-based devices. Developers currently using products in the DSP56300 family will find the SC140 instruction set and programming techniques already familiar as they migrate their application to the SC140 multi-ALU architecture. The SC140 core architecture includes features that influence how DSP56300 software must be modified to operate efficiently on the SC140 core.

Note: The SC140 and SC1400 cores are functionally identical, and the information in this document applies to both cores. For simplicity, the SC140 core is referenced throughout this application note.

This application note highlights the SC140 architectural features that affect software and describes how to use these features when converting DSP56300 code, especially assembly code. C code compiles efficiently on the SC140 core. Although you can convert assembly code from DSP56300 to SC140, it may be preferable to convert DSP56300 code to C code and then program only the critical sections in StarCore™ assembly code, if necessary. This process is discussed in **Section 3**. Peripheral differences between the DSP56300 and SC140-based families are not discussed because they are device-specific. Finally, differences between the Suite56™ and CodeWarrior® tools are briefly considered.

CONTENTS

freescale™ semiconductor

1 Core Architectures Compared

Both the DSP56300 core and the SC140 core are high-performance DSP architectures. Devices based on these cores operate at high frequencies with low power and provide dedicated arithmetic support for DSP applications. This application note focuses on porting code from any DSP56300 device to a device based on the SC140 core. Therefore, the main interest is the core and memory expansion areas of these devices.

1.1 16 Bits Versus 24 Bits

The DSP56300 core is a 24-bit architecture, so arithmetic operations process 24-bit operands and produce 24-bit results. The SC140 core is a 16-bit architecture that operates on 16-bit operands. This change in operand size affects memory requirements and precision. A DSP56300 *word* is 24 bits, but an SC140 word is 16 bits. Both cores support double-precision operations. The term *long* refers to 48 bits on the DSP56300 core and 32 bits on the SC140 core. Both the DSP56300 and SC140 accumulators have eight extension bits. The DSP56300 core includes an option for operating in 16-bit compatibility mode. In this mode, there is no mathematical effect in changing to the 16-bit architecture of the SC140 core. The same holds true for a conversion from the DSP56F800 core to the SC140 core. The DSP56F800 is a 16-bit core with the same level of precision as the SC140 core. However, if the default 24-bit architecture of the DSP56300 core is used, you must remember that SC140 is a 16-bit architecture. Limiting, rounding, and overflow occurs on 16 bits rather than 24 bits, and scaling may be required where it was not on the DSP56300. Scaling mode for the SC140 core is set in the Status Register (SR).

If all 24 bits of precision are required for an application running on the DSP56300 core, the double-precision capabilities of the SC140 core can be used to achieve the required precision. If more than 16-bit precision is required, the SC140 core can execute addition, subtraction, and logical operations on 32 bits in a single cycle. **Example 1** shows code for these single-cycle 32-bit addition and subtraction operations in comparison with equivalent 24-bit DSP56300 code. Notice that there is no performance or precision degradation for these operations on the SC140 core.

Example 1. Double-Precision Addition and Subtraction Comparison

```
DSP56300 24-bit precision code:
add x1,a
sub y1,b
eor y0,b
```

```
SC140 32-bit precision code:
add d0,d1,d2
sub d3,d4,d5
eor d6,d7
```

The *SC140 Core Reference Manual* describes how the SC140 core can emulate double and mixed precision multiplies and multiply-accumulate operations using the `mpy<su,us,uu>`, `mac<su,us,uu>`, and `dmac<ss,su>` instructions. The `s` refers to a signed operand and `u` refers to an unsigned operand. The double-precision multiplication operation (32-bit × 32-bit) consumes four core cycles for one ALU, compared with a single-cycle multiplication for single-precision (see **Example 2**). If all 64 bits of the result are needed, additional transfers can be added, as described in the *SC140 Core Reference Manual*. Since the focus here is on achieving the 24-bit level of precision, the transfers are not included.

Example 2. Double-Precision Multiplication Comparison

```
DSP56300 24-bit precision code:
mpy x0,y0,a
```

```
SC140 32-bit precision code:
mpyuu  d0,d1,d2
dmacsu d0,d1,d2
macus  d0,d1,d2
dmacss d0,d1,d2
```

Mixed-precision operations (32-bit × 16-bit) consume two core cycles for one ALU, as shown in **Example 3**.

Example 3. Mixed-Precision Multiplication Comparison

```
DSP56300 24-bit precision code:
mpy x0,y0,a
```

```
SC140 mixed precision code:
mpysu  d0,d1,d2
dmacss d0,d1,d2
```

For applications exploiting instruction-level parallelism by implementing multiple operations simultaneously on the four SC140 ALUs, the penalty for using double-precision arithmetic can be reduced, as shown in **Example 4**. The DSP56300 core requires additional load/store operations that are not shown to feed the four multiplication operations. These operations primarily use parallel moves, but require at least one additional cycle to store an accumulator result before the previous one is overwritten. The SC140 core requires four cycles to implement this double-precision multiplication operation. Each line of text in **Example 4** corresponds to one cycle of core execution time. This example shows that multiple high-precision operations in a DSP56300 design, such as the four shown here, can be parallelized on the SC140 core with no cycle degradation. Four high-precision multiplies on the DSP56300 core consume a minimum of four cycles. The same holds true for the SC140 core.

Example 4. Double-Precision Parallelism

```
DSP56300 24-bit precision:
mpy x0,y0,a   ; result 1
mpy x1,y1,b   ; result 2
mpy x0,y0,a   ; result 3
mpy x1,y1,b   ; result 4
```

```
SC140 32-bit precision:
mpyuu d0,d1,d2   mpyuu d3,d4,d5   mpyuu d6,d7,d8   mpyuu d9,d
dmacsu d0,d1,d2  dmacsu d3,d4,d5  dmacsu d6,d7,d8  dmacsu d9,d
macus  d0,d1,d2  macus  d3,d4,d5  macus  d6,d7,d8  macus  d9,d
dmacss d0,d1,d2  dmacss d3,d4,d5  dmacss d6,d7,d8  dmacss d9,d
; four results in d2, d5, d8, d11
```

See **Section 3** for information on C programming support for these double-precision arithmetic operations. The application note, AN2208, *Implementing a Double-Precision (32-bit) Complex FIR Filter,* provides additional examples of double-precision multiplication on the SC140 core.

Division is an iterative process in which the number of cycles is based on the amount of precision required. The divide iteration on the SC140 core is based on the 16-bit architecture. When more than 16-bits of precision are needed for a division operation on the SC140 core, a user-defined *N*-bit division routine is required. The cycle requirement for this operation varies according to the routine selected.

1.2 Endianness and Bit Ordering

The SC140 core is a big-endian core in the currently-available MSCxxxx devices. In contrast, the DSP56300 core is a little-endian core. This difference has little impact on programming because the assemblers and compilers arrange the opcodes according to the processor format. Also, because the DSP56300 core is word-addressable, the byte ordering is not affected by endianness as it is on the byte-addressable SC140 core. The documentation for the two cores does use different bit ordering conventions. Consider the hexadecimal value 0x12345678. The DSP56300 considers the right-most bit in the 0x8 nibble to be the least significant bit and it is numbered as bit 0 in the DSP563xx manuals. The SC140 documentation, such as the *SC140 Core Reference Manual*, is written for the SC140 implemented as a little-endian core, so the documentation is numbered like the DSP56300 documentation. However, the device-specific reference manuals may number the bits with bit 0 in the most-significant bit position. This numbering can initially be confusing when you look at register settings, especially of peripheral registers, and count which bit is being programmed.

Although endianness differences do not cause big programming differences between the devices, it is important to know that the SC140 is implemented as a big-endian core in the MSCxxxx devices. Any data sent to the device should be sent in big-endian ordering. Otherwise, incorrect results are calculated when the core retrieves the data

for processing. The endianness affects byte ordering between the two devices. **Example 5** shows the difference in how data bytes (for a 32-bit value) are stored in a big-endian device versus a little-endian device and the addressing difference compared with the DSP56300 core.

Example 5. Endianness and Addressing Comparison

```
Store the value 0xCAFEFOOD at address 0x00001234

DSP56300 'Little' Endian:
0x001234      0xCAFEFO

SC140 Little Endian:
0x00001234    0xOD
0x00001235    0xFO
0x00001236    0xFE
0x00001237    0xCA

SC140 Big Endian:
0x00001234    0xCA
0x00001235    0xFE
0x00001236    0xFO
0x00001237    0xOD
```

Note that future devices may implement the SC140 as a little-endian core, so it is best to check the specific device reference manual to ensure the endianness implementation.

1.3 Memory Organization

Unlike the DSP56300 core, which has separate X, Y, and P (program) memory buses/banks, the SC140 core has a unified memory architecture that holds data and program information in a single block of memory with multiple bus ports. This unified memory architecture gives you more flexibility in placing application information.

1.3.1 Byte-Addressable Versus Word-Addressable

The 24-bit architecture of the DSP56300 core implements all calculations on word operands. Since a word is 24 bits (or 3 bytes), the memory is word-addressable. That is, each address corresponds to one word in memory. Address $0 corresponds to the first three bytes of memory, address $1 corresponds to the next three bytes, and so on. Although the SC140 is a 16-bit architecture, it includes flexibility to operate on both 8-bit and multi-byte operands, with corresponding flexibility in memory addressing. The SC140 core is a byte-addressable architecture, so address 0x0 corresponds to one byte, address 0x1 corresponds to the next byte, and so on.

Note: The DSP56300 and SC140 compilers use different conventions for representing hexadecimal numbers. The DSP56300 compiler precedes a hexadecimal number with a $ symbol; the SC140 compiler precedes the number with a 0x notation.

The SC140 address arithmetic unit (AAU) updates pointers based on the size of data. If a pointer register is programmed for post-increment when a 2-byte operand is accessed, it increments the pointer by 2 (that is, `move.w xx,(r0)+`). If it accesses a byte operand, the pointer increments by just 1 (that is, `move.b xx,(r0)+`). Therefore, the address offsets in DSP56300 code may not need to change when converted to SC140 code, but it is important that you understand the difference in case you see strange addressing that needs to be updated.

1.3.2 Data Alignment

Because the DSP56300 core is word-addressable, data is aligned on 24-bit boundaries. However, since the SC140 core can operate on various sizes of data and uses its variety of AAU instructions to select enough data to feed all four ALUs, data alignment becomes very important in the SC140 core. When code is converted from DSP56300 to SC140, you must ensure that data is aligned on the appropriate boundary for the operations performed on the data. The 16- and 32-bit operands must be aligned on 16 and 32-bit boundaries (addresses ending in $0, $2, $4, ... for 16-bit and $0, $4, $8, ... for 32-bit).

1.3.3 Memory Grouping

For the DSP56300 core, you must select the memory configuration to place the appropriate amount of memory in each bank. The SC140 core has a unified memory map, and no memory configuration selection is programmed. The size and design of the memory available to the SC140 core is device-dependent, so you should be aware of memory architecture in the device you are programming. Refer to the SC140-based device reference manuals, such as the *MSC8101 Reference Manual* or the *MSC8102 Reference Manual,* for details on memory grouping and simultaneous memory accesses. Contention can occur between the core program and data accesses to the same sub-group of memory, so some memory planning is still required even though the SC140 core does not use an explicit memory configuration switch. The compiler can examine all memory groups and place data and program information to avoid contention. If you decide to bypass the compiler placement algorithms, then you must place the information to avoid contention in order to get optimum performance.

1.4 Bus Structure

Figure 1 shows the SC140 core architecture, and **Figure 2** shows the DSP56321 block diagram.

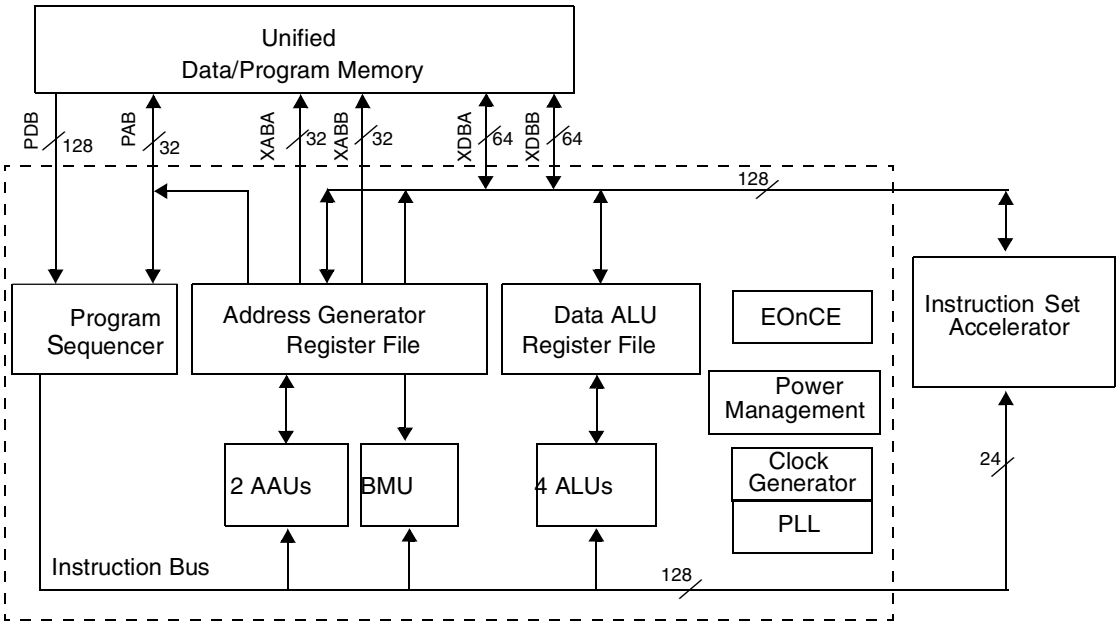


Figure 1. SC140 Core Block Diagram

The DSP56300 bus structure differs from the SC140 bus structure, as a comparison of **Figure 1** and **Figure 2** shows. The DSP56300 core has 24-bit wide X address and data buses, Y address and data buses, program address and data buses, and DMA address and data buses. In addition, a switch on the address and data buses gives the DSP56300 core direct access to external memory. In contrast, the SC140 address buses are 32-bits wide, the data buses are 64-bits wide, and the program bus is 128-bits wide. The SC140 program data bus must be wide enough to retrieve fetch sets that include more than one instruction word from memory. None of the current SC140-based devices give the SC140 core direct access to the DMA controller. Instead, the DMA controller and any external memories reside on a separate bus on the other side of a switch from the core. In SC140-based devices the DMA controller has direct access to the unified memory through its own dedicated port.

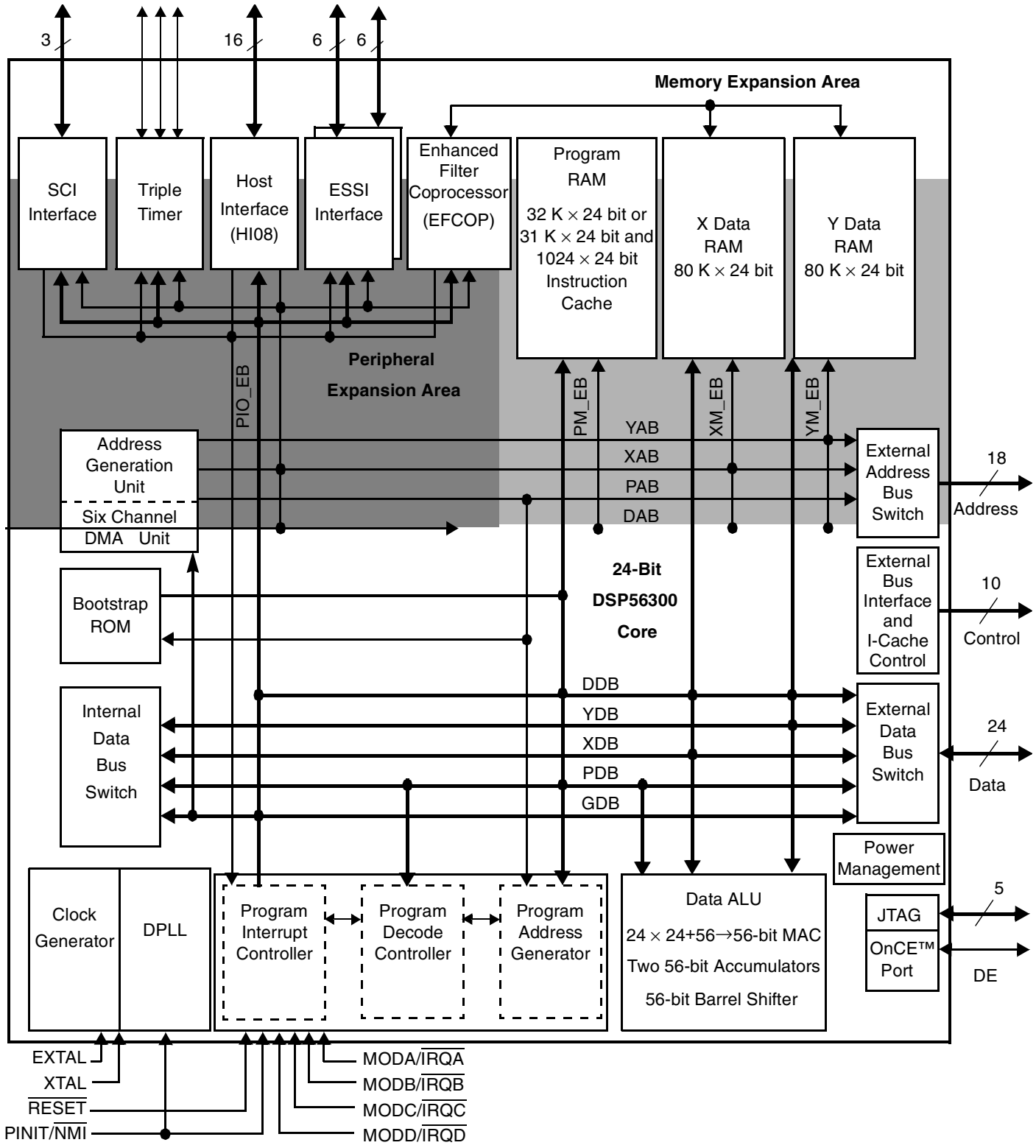


Figure 2. DSP56321 Block Diagram

1.5 Processing Units

The SC140 core has three types of processing units: arithmetic logic unit (ALU), address arithmetic unit (AAU) and bit mask unit (BMU). The DSP56300 core has an ALU and address generation unit (AGU) but does not include a BMU. The DSP56300 can combine a single ALU calculation with two address calculations in a single cycle of program execution. The SC140 can combine four ALU calculations with two address calculations in a single cycle. This section describes the differences between these units on the SC140 and the DSP56300 cores.

1.5.1 ALU

Both the DSP56300 and SC140 ALUs include a multiply-accumulate unit, a bit-field unit, and a data shifter/limiter. The heart of both ALUs is the single-cycle multiply-accumulate unit. Because both devices provide this unit, the programming modifications for porting algorithms from the DSP56300 to the SC140 core lie primarily in data addressing and data register usage. The DSP56300 and SC140 ALUs differ in two primary ways: number of ALUs and number/format of ALU data registers. The SC140 core contains four ALUs so that up to four ALU instructions can execute in a single cycle. In contrast, the DSP56300 core executes only a single ALU instruction each cycle.

For assembly programming, the multi-ALU capability of the SC140 can result in code that is denser and more challenging to follow. You can attempt to fit all ALU instructions to execute in a group on a single line of code, or you can use square brackets to group these multiple instructions. The left square bracket goes before the first instruction, and the right square bracket follows the last instruction to be executed in a group. The line of code does not become too wide and difficult to read, and the grouping is clear. **Example 6** shows these two methods of programming.

Functionally, the DSP56300 and SC140 ALUs use their associated registers differently. The DSP56300 has four input registers, X0/X1/Y0/Y1. The values from these registers are input to the multiplier and then to either accumulator A or B. The value of the accumulator is placed back on the bus for returning the value to memory. The SC140 ALU has 16 data registers that act not only as input to the ALU processing but also as the accumulator registers for ALU operations. Therefore, these registers are the source registers for placing final calculations back into memory. In contrast with the X,Y, and accumulator registers of the DSP56300, the SC140 core does not have ALU registers with different functionality. All D*n* registers of the SC140 ALU are interchangeable and can be used in all ALU instructions. Excluding the `mac` instruction, most DSP56300 instructions are two-operand instructions. Because the SC140 core has an increased number of data registers, many frequently-used instructions are available as three-operand instructions so an application can preserve the values in the source registers.

Example 6. ALU Usage Comparison

```
DSP56300 code:
mac x0,y0,a
```

```
SC140 code:
mac d0,d4,d8   mac d1,d5,d9   mac d2,d6,d10   mac d3,d7,d11
```

```
OR:
[
mac d0,d4,d8
mac d1,d5,d9
mac d2,d6,d10
mac d3,d7,d11
]
```

In addition, the DSP56300 core combines X0 and X1 into a single register when it works on double-word (48-bit) operands. The 16 data registers shared among the four ALUs of the SC140 core provide double-word (32-bit) arithmetic. They can be accessed as 16-bit registers by specifying the low (d0.l) or high (d0.h) portions of the

registers, or they can be referenced as a single 32-bit register (d0). While X0 is a 24-bit register in the DSP56300, D0 of the SC140 is a 40-bit register consisting of a 16-bit low portion, 16-bit high portion, and an 8-bit extension. Instructions are available to access portions of the SC140 data registers. The saturation instructions treat the data as either 16-bit fractional (.f) or 32-bit long (.l) and saturate accordingly. Also, data can be sign or zero-extended to byte (.b), word (.w), or double-word (.l). The SC140 AGU provides many different types of move operations to transfer data in and out of the different portions of the ALU data registers.

1.5.2 Address Arithmetic Unit (AAU)

In **Figure 2** on page 6, the DSP56300 core calls the AAU as an address generation unit. The AAUs of the DSP56300 and SC140 cores are similar, so there are not many programming differences. However, the differences that exist must be understood.

The AAUs of both the DSP56300 and the SC140 cores have two address arithmetic units. Both also have eight address registers, R[0–7]. The DSP56300 core restricts which address registers can be used with which address arithmetic unit. The SC140 core makes all address registers accessible to both address arithmetic units. Each DSP56300 address register has its own dedicated address modifier register (M[0–7] and N[0–7]). In contrast, the SC140 core provides four of each type of address modifier registers (M[0–3], N[0–3]) that can be associated with any address register by programming the Modifier Control Register (MCTL). Stack pointer differences are discussed in **Section 1.5.4**.

In addition to the original eight address registers, the SC140 core provides eight base registers (B[0–7]) to specify the base address of modulo buffers when modulo addressing is used for one of the lower eight address registers (R[0–7]). If modulo addressing is not enabled to use a particular base register, that base register can be used as an additional address register. Therefore, the B[0–7] base registers can also be referred to as the R[8–15] address registers, significantly increasing the number of available address registers.

You may wonder how the SC140 core can increase its ALU functionality four-fold over the DSP56300 core, as shown in **Example 7**, and yet retain only two address arithmetic units. The answer lies in the size of its buses. The DSP56300 code shows the two parallel AGU moves that can occur in parallel with an ALU instruction. The SC140 code shows the two parallel AAU moves that can execute in parallel with four ALU instructions. Notice that the SC140 core can fill eight registers in a single cycle.

Example 7. AAU Code Comparison

DSP56300 code:	SC140 code:
`move x:(r0)+,y0  move y:(r4)+,y0`	`move.4w (r0)+,d0:d1:d2:d3  move.4w (r1)+,d4:d5:d6:d7`

1.5.3 Bit Mask Unit (BMU)

The BMU is actually part of the SC140 address generation unit, which includes the two AAUs described in the previous section and one BMU. This is important to remember when grouping instructions because BMU instructions take the place of one AAU instruction in an execution group.

When you are completing an initial port of DSP56300 code to SC140 code, the bit mask unit will probably be used only for the logical instructions (AND, NOT, EOR, and OR). There is no explicit support for the remaining bit-mask unit operations in the DSP56300 core. However, when you optimize the ported SC140 code, you should consider how BMU instructions may improve efficiency in decision-making and resource sharing.

1.5.4 Program Control

The DSP56300 core uses a program control unit (PCU) to fetch new instructions, handle exceptions, and implement the pipeline. The SC140 core uses a program sequencer (PSeq) to perform these functions. Their purpose is the same in each core, but there are significant differences.

In both the DSP56300 and SC140 core architectures, most instructions in DSP algorithms execute in one cycle. The DSP56300 core has a seven-stage interlocked pipeline, and the SC140 core has a five-stage pipeline with no interlocks between the stages. The absence of interlocks in the SC140 pipeline provides more programming flexibility and more awareness of code optimization locations. The SC140 code development tools provide warnings if there is a possibility of incorrect code execution due to code ordering and the pipeline. Therefore, you are alerted when the code can be rearranged for more optimized performance—rather than having the device transparently add `NOPs`, as the DSP56300 core does.

The first two stages of the SC140 pipeline are pre-fetch and fetch. This is similar to the DSP56300 core. However, the SC140 core fetches eight instruction words (128 bits) in contrast to the DSP56300 core, which fetches only a single instruction word (24 bits). An SC140 fetch set is not the same as an execution set. The SC140 core fetches the maximum possible instruction words each cycle. The program sequencer detects which instructions are grouped into an execution set. On the next stage of the pipeline, the execution set is dispatched to the execution units. The last two stages of the pipeline are address generation and execution. The DSP56300 core has two pipeline stages for each of these tasks.

In the dispatch stage of the pipeline, the SC140 program sequencer determines the grouping of each execution set. The SC140 core has a variable length execution set (VLES) architecture, so each execution set can have from one to six instructions, depending on the instruction type. There are four classifications of instruction type:

- Type 1 includes basic DALU and move instructions that are frequently used.
- Type 2 includes additional DALU, move, and AGU arithmetic instructions.
- Type 3 are two-word and three-word DALU, move, and AGU arithmetic instructions.
- Type 4 includes all other instructions such as change-of flow instructions.

Chapter 6 of the *SC140 Core Reference Manual* describes the restrictions on how instruction types can be grouped for this architecture.

Both the DSP56300 and the SC140 cores support hardware loops. The DSP56300 core includes a single set of hardware loop registers and allows for as much loop nesting as there is memory available to store the status of the loop registers. The SC140 core includes four sets of hardware loop registers that can nest within each other, with a maximum of four loop nesting levels. Details on the coding differences between hardware loops on the DSP56300 and SC140 core are discussed in **Section 2.2**.

The DSP56300 core has a hardware stack that can be extended into X or Y data memory. The SC140 core has two software stack pointers: normal (NSP) and exception (ESP). The normal stack is used by tasks when the SC140 core is in the normal mode of processing. The exception stack is used by interrupts and/or an operating system when the SC140 core is in exception mode, indicated by the EXP bit in the status register. Stacks for both cores store the Program Counter (PC) and Status Register (SR) values when entering subroutines. These values are restored automatically from the stack when returning from a subroutine. To retain the performance benefit of a hardware stack while providing the flexibility of a software stack, the SC140 core includes a register called the Return Address Stack (RAS) that is updated with the return address when a subroutine is called. Upon returning from the routine, the return address is taken from the RAS register instead of the stack, although the SC140 core

also updates the stack. When subroutine calls are nested, the RAS is valid for the leaf routine. Instructions for accessing the SC140 stack are provided in **Section 2.3**. It is important to initialize both values and not to forget the ESP when you are porting code from the DSP56300 core.

2 Assembly-Level Functional Differences

At the assembly language level, architectural and design differences can have an impact on code porting between the DSP56300 core and the SC140 core. This section provides tips on porting assembly code directly from a DSP56300 device to assembly on an SC140 device.

2.1 Addressing Modes

As described in **Section 1.5.2**, there are differences between the DSP56300 and SC140 AAU registers and therefore some differences in addressing modes. Instead of the eight modifier (M*n*) and eight offset (N*n*) registers in the DSP56300, the SC140 includes four modifier and four offset registers.

The offset registers of both cores allow an address register value either to be combined with an offset register value to calculate a variable address or to update an address register during or after an access. The SC140 offset registers can increment or decrement address registers in register update calculations. Although there are fewer offset registers on the SC140 core, they can be used with any of the address registers. In the DSP56300 core, the N0 register must be used to update or offset R0. In the SC140 core, any of the four offset registers can be used as an offset with R0. There is no restriction on which offset register is used with an address register. In addition, unused SC140 address registers can be employed as offset registers in some addressing modes.

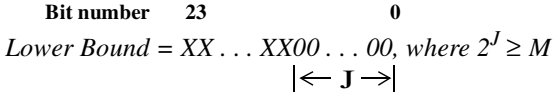
Since the SC140 core is not word-addressable and allows access widths of varying sizes, the SC140 AAU automatically shifts the offset value to align with the access width of a given transaction. The programmer does not need to vary the offset size for each access size.

The Indexed By Offset, (Rn + Nn) addressing mode differs between the DSP56300 core and the SC140 core. The DSP56300 address register, R*n*, can be indexed by its associated offset register, N*n*. The SC140 core allows indexing by offset register N0 or a separate address register (R[0–7]). N1, N2, and N3 are not usable in this particular addressing mode. While the SC140 allows unused address registers to be repurposed for this indexing, you must ensure that this change is taken into account and that N[1–3] are not selected for this use when code is converted from DSP56300 code.

The SC140 core does not support the predecrement by 1, -(Rn) addressing mode. It does support Postdecrement by Offset N*n*, although it does not explicitly list this mode in its address modes list. The SC140 core treats the offset register value as a signed value, so a negative offset value used in Post-Increment by Offset, (R*n*)+N*n* mode effectively implements a post-decrement by offset transaction.

Both the SC140 and DSP56300 cores support modulo addressing. However, the SC140 core simplifies the placement and planning for modulo buffers. The DSP56300 core requires the following steps to set up the modulo registers.

1. Set the modifier register to the size of the buffer minus 1, $Mn = M - 1$.
2. Calculate the lower bound of the buffer.



3. Calculate the upper bound of the buffer:

$$\begin{array}{l}\textbf{Bit number} \quad 23 \qquad\qquad 0 \\ \textit{Upper Bound} = XX \ldots XX00 \ldots 00 + M - 1. \\ \qquad\qquad\quad \left|\leftarrow \mathbf{J} \rightarrow\right|\end{array}$$

4. Set the address register, $LowerBound \leq AddressRegisterRn \leq UpperBound$.
5. Set the offset register, $Nn = increment \leq M$.

The DSP56300 core does not include an explicit buffer for defining the base address of the modulo buffer, so the core calculates the base address using the size of the buffer and location of the address register. For the calculation to work, the buffer must be placed at a location with the correct number of zeroes in the lower part of the address. This process can easily result in errors if the buffer is placed incorrectly.

The SC140 core includes explicit base registers in its AAU, which greatly simplify the process of locating a modulo buffer and thereby simplify memory management. Each address register, R[0–7], has an associated base register, B[0–7]. If an address register is assigned to point to a modulo buffer, its associated base register must be programmed to the base address of the buffer. Because the base address is explicitly set in a register, the requirement for a certain number of zeros in the lower portion of the buffer base address is no longer necessary. The SC140 core does not provide a separate buffer size register, M*n*, for each address register. Often, buffer sizes are common among buffers, so the SC140 core provides four M*n* registers for use by multiple active buffers simultaneously. The process for initializing modulo buffers on the SC140 core is as follows:

1. Program R*n* to an address within the buffer.
2. Program modulus size minus 1 to the M*n* register.
3. Program base register, B*n*, to the modulo buffer base address.
4. Program the MCTL to indicate which address register(s), R*n*, points to a modulo buffer and which modulo size register, M*n*, should be used for the buffer(s).

Example 8 shows SC140 code for setting up two modulo buffers of the same size.

Example 8. SC140 Modulo Buffer Initialization

```
move #ModBuff1Addr,r1     move #ModBuff2Addr,r3
move #BuffSize-1,m0
move #ModBuff1Base,b1     move #ModBuff2Base,b3
move.l #$00008080,MCTL
```

Address register 1, r1, and address register 3, r3, each point to a modulo buffer. Both buffers are the same size. Base address register 1, b1, stores the base address of buffer 1. Base address register 3, b3, stores the address of buffer 2. The MCTL is programmed so that r1 uses modulo addressing with address modifier register m0. It also programs r3 to use modulo addressing with address modifier register m0.

2.2 Hardware Loops

Both the DSP56300 and SC140 cores support hardware loops. The software syntax for initiating a loop differs slightly between the two cores. The DSP56300 has loop count (LC) and loop address (LA) registers that are initialized at the start of a loop. When loops are nested, the values of LC and LA are placed on the system stack as program flow enters a new loop. Up to seven DO loops can be nested. If the system stack extension is enabled, the number of nested DO loops is limited only by the size of the extension provided by the application programmer. The SC140 core provides four loop start address registers (SA[0–3]), four loop counter registers (LC[0–3]), and five loop flags in the status register. This means that the SC140 core can support up to four nested hardware loops.

Note: When loops are nested on the SC140 core, keep in mind that a loop can be nested only within a loop that has a lower index (loop 3 inside of loop 2/1/0, loop 2 inside loop 1/0, loop 1 inside of loop 0).

Example 9 and **Example 10** show loop programming syntax for the DSP56300 and SC140 cores. The DSP56300 core requires a statement at the start of the loop designating the number of loop iterations and the address of the end of the loop. The loop starts immediately after this statement.

Example 9. DSP56300 Hardware Loop Syntax

```
    do _loop_cnt,_StopLoop
    mac x0,y0,a     x:(r0)+,x0     y:(r4)+,y0
    mac x1,y1,b     x:(r1)+,x1     y:(r5)+,y1
_StopLoop
```

In contrast, the SC140 core requires a statement to program the loop count and loop address values for a specific loop. In **Example 10**, the starting address of the loop and the number of loop iterations is programmed to the LC and SA registers. `dosetup<n>` places the loop start address in the start address register (SA). `doen<n>` places the number of loop iterations in the Loop Count Register (LC). Because they are dedicated to each loop, the loop registers can be initialized at the beginning of an application and invoked when needed. The `loopstart0` and `loopend0` directives are assembler directives to indicate the start and end of the loop to the assembler. These directives do not consume core clock cycles.

Example 10. SC140 Hardware Loop Syntax

```
  dosetup0 _StartLoop0
  doen0 #loop0_iterations
  ; more code can go here
  loopstart0
_StartLoop0
  mac d0,d1,d2    add d5,d6,d4   inc d6   move.2w (r0)+,d0:d1  move.2w (r1)+,d
  sub d7,d8,d9    add d2,d4,d4            move.w (r3)+n1,d7    move.w (r4)+n2,
                                          move.w d4,(r2)+      move.w d9,(r5)+
  loopend0
_EndLoop0
```

There are multiple options for converting a loop from DSP56300 assembly code to an SC140 loop:

- Perform four operations simultaneously.
- Process four of the operations in a single cycle and divide the number of loop iterations by four.

There are multiple methods for arranging loops to optimize their performance in the SC140 multi-ALU architecture. These methods are detailed in application notes available on the SC140-based device product summary web sites. Also, you can maximize loop efficiency even when programming in C. AN2009, *Introduction to the StarCore SC140 Tools: An Approach in Nine Exercises*, is a helpful introduction to programming SC140 using the CodeWarrior tools from Metrowerks. AN2266, *Developing Optimized Code for Both Size and Speed on the SC140*, is another useful application note for developing optimized code. Examples in these application notes describe loop unrolling, split summation, and multi-sample techniques for optimizing loops in a multi-ALU device.

2.2.1 Short Loops

The DSP56300 core includes a `rep` instruction for an even faster hardware loop configuration called a repeat loop, which executes very rapidly because it allows only a single instruction to repeat. This instruction is fetched once at the beginning of the loop, and no additional fetches are required for the loop to execute to completion. The disadvantage of this loop is that it is not interruptible, so if an application requires program flow to respond to interrupts immediately and a repeat loop with a large loop count is executing, there can be latency issues.

The SC140 core addresses this drawback with a mechanism called a short loop. The short loop provides the same type of performance gain since its size is restrained to one or two execution sets so that program fetches are not required to execute the loop to completion. However, the short loop is interruptible. **Figure 11** shows the syntax for short loops. The DSP56300 code on the left shows the single-cycle execution allowed in the repeat loop. The SC140 code on the right shows an example of a short loop with two execution sets.

Example 11. Short Loop Syntax Comparison

```
rep #N-1
mac x0,y0,a    x(r0)+,x0    y:(r4)+,y0
```

```
doensh0 #$10
loopstart0
mac d0,d1,d2    move.w (r0)+,d0
add d5,d6,d4    move.w (r1)+,d5
loopend0
```

2.2.2 Additional Looping Instructions

Both the DSP56300 and SC140 cores provide additional instructions for loop support. On the DSP56300 are BRKcc, DO FOREVER, DOR, DOR FOREVER, and ENDDO. DOR is a PC-relative hardware loop. The SC140 does not include support for PC-relative hardware loops. It also does not include infinite loops such as the DO FOREVER loop. These DSP56300 loops must be converted to standard hardware loops on the SC140 core.

The additional SC140 looping instructions are SKIPLS, CONT, CONTD, and BREAK. BREAK provides the same functionality as BRKcc when it is combined with an IFc instruction. SKIPLS provides new functionality for jumping the program flow to a defined label if the loop count is not a positive value to avoid the extra cycles of initiating a loop. When loop count is greater than 1, the CONT instruction allows the program flow to bypass any remaining portion of a loop and return to the beginning of the loop. If loop count is less than or equal to 1, the program flow can exit the hardware loop by jumping to a designated label. CONTD has the same purpose as CONT but includes a delay slot. Delay slots are described in **Section 2.3.2**. These new SC140 looping instructions provide additional functionality to increase efficiency in assembly-coded algorithms.

`falign` is not a looping instruction, but an assembler directive that significantly affects loop efficiency on the SC140 core. When loops are converted from DSP56300 to SC140 code, this directive should be placed immediately before the SC140 loop. To optimize how the program control unit fetches a VLES at the start of a loop, the loop must be aligned so that the VLES is aligned on a loop boundary. Fewer fetches can occur on each loop iteration and thereby improve performance if the loop is aligned. `falign` adds `nops` in front of the loop to align the loop. An alternative to `falign` is to check the loop alignment and arrange instructions before the loop so that the loop is aligned in memory on a fetch set boundary.

2.2.3 Looping Restrictions

Both the DSP56300 and the SC140 cores restrict nesting and change-of-flow instruction placement in loops. The restrictions are similar between the two cores, but the SC140 core adds a few changes because of its VLES architecture. If you are programming in C, the compiler generates code that avoids any restrictions. However, when assembly code is converted, keep the following differences in mind:

- Conditional branches or jumps (Bcc/Jcc) are not allowed in the last four execution sets of a long loop (LA-3 to LA).
- The DOEN/DOENSH instruction cannot be placed between the DOEN/DOENSH instruction and the start address of a second loop.
- If a short loop is nested inside another loop, the last address of the short loop should not appear in the last two execution sets of the outer loop (LA, LA-1).
- A move-like instruction that changes the loop flags in the status register is not allowed in the execution set immediately before a DOEN/DOENSH instruction.
- Only loop instructions should update the status register during active loops.

In addition, certain operations are not allowed in short loops. Refer to the *SC140 Core Reference Manual* for details on these restrictions. For converting code from the DSP56300, these operations are not an issue because they would not occur in a repeat loop, but they should be considered when you are writing new assembly code for the SC140 core.

2.3 Instruction Set Differences

Although the DSP56300 and SC140 instruction sets are similar, there are differences due to new architectural features of the SC140 core. Table 3, “Instruction Set Comparison,” on page 21, is a useful reference when you are converting assembly code between the cores. One significant difference between the cores lies in the capabilities of their AGUs. The DSP56300 AGU is used only for moving data around the device. The SC140 AGU has expanded arithmetic capabilities for modifying data in memory and performing arithmetic operations on AGU registers. See Table 4, “SC140 AGU Arithmetic Instructions,” on page 24.

The SC140 core supports 5-bit or 16-bit immediate values for its immediate instructions. The DSP56300 core generally provides 6-bit and 16-bit immediate options on its immediate instructions. If a 6-bit immediate value is used in the DSP56300 code, the converted code should either use a 16-bit immediate value in the SC140 assembly or the line should be modified so it can use a 5-bit immediate value.

To access the stack in the DSP56300, `move` instructions are used with the stack as the source or destination. The SC140 has a normal stack pointer and an exception stack pointer. The SC140 instruction set provides explicit instruction support for accessing the stack. The SC140 stack support instructions are listed in **Table 1**. Two of these instructions can be implemented in parallel if they follow the guidelines described in chapter 5 of the *SC140 Core Reference Manual*.

Table 1. SC140 Stack Support Instructions

Instruction	Description
POP	Pop a register from the software stack
POPN	Pop a register from the software stack using the normal stack pointer
PUSH	Push a register onto the software stack
PUSHN	Push a register onto the software stack using the normal stack pointer
TFRA	Move the other stack pointer to/from a register

SC140 ALU support for integer arithmetic is provided in a group of arithmetic instructions with the “i” prefix. These instructions perform calculations with the decimal point immediately to the right of the least significant bit of the lower portion of the data register. These instructions sign-extend or zero-extend the data register based on whether the arithmetic is signed or unsigned. Data transfer instructions are available to move data into the low portion of the register to support this arithmetic mode. Data transfer instructions are also available to move data in and out of the high portion of data registers for fractional arithmetic. Fractional arithmetic instructions perform arithmetic calculations with the decimal point immediately to the right of the most significant bit of the high portion of the register.

The DSP56300 data movement operations are based on its 24-bit architecture. All moves are related to 24 or 48 bit operands. Because the SC140 core is byte-addressable and feeding four ALUs, it includes more flexibility in operand sizes and quantity of data to be moved. For 16-bit operands, it uses the .w and .f suffixes. The .w suffix indicates that a word is being moved. When a word is moved, the 16 bits are moved to/from the lower 16 bits of the operand and the upper portion of the register is sign-extended or zero-extended, depending on whether it is a signed or unsigned move. The .f suffix indicates that the data is fractional. Fractional data is moved to/from the upper 16 bits of the operand. The lower portion of the register is cleared. A .l suffix is also available for moving 32-bit data into the data registers. Because the data buses are wider on the SC140 core than on the DSP56300 core, it can use multi-operand moves, transferring two fractionals or words, four fractionals or words, or two longs in and out of memory using a single AAU `move` instruction with the correct suffix.

The SC140 core includes expanded instruction support for Viterbi decoding kernels. In addition to the VSL instruction also available in the DSP56300, the SC140 includes a MAX2VIT instruction to update two Viterbi flags (VFs) in the status register. The application note, ANSC140VIT/D, *How to Implement a Viterbi Decode on the StarCore SC140*, explains how to use these instructions as well as the ADD2 and SUB2 instructions in the context of Viterbi decoding.

Two new instructions are available on the SC140 core for managing interrupts: `ei` (enable interrupts) and `di` (disable interrupts). These instructions affect any maskable interrupts. In addition, their effect is immediate—even exception-causing instructions in the same execution set with the `ei`/`di` are enabled or disabled by execution of these interrupt instructions. These instructions can be used to protect code that should not be interrupted.

2.3.1 Conditional Operations

The DSP56300 status register includes eight condition codes in its condition code register (CCR). The SC140 core includes a single condition bit in the status register, the true (T) bit, which changes how conditional operations are programmed. For conditional change of flow, the DSP56300 core provides an extensive list of possible `Bcc` and `Jcc` instructions based on each condition code. These instructions rely on a previous compare instruction to set the appropriate status register condition bit. Then the change-of-flow instruction checks the status register bit for the specified condition and executes a change of flow based on the result of the test. The SC140 core provides test (`TSTcc`) and compare (`CMPcc`) instructions for testing a given condition. The difference is that the SC140 core includes the condition being checked for in the comparison instruction rather than in the change-of-flow instruction. If the comparison condition is true, the T bit in the status register is set. If it is false, the bit is cleared. Conditional branches and jumps then become a simple branch-true/false (`BT`/`BF`) or jump-true/false (`JT`/`JF`). Also, the SC140 core provides comparison instructions for both ALU and AAU accumulators/registers. **Example 12** compares how the two devices handle conditional change of flow.

Example 12. Conditional Change-of-Flow Comparison

DSP56300 code:	SC140 code:
cmp x0,a	cmphi d0,d1
bgt BRANCH_LOCATION	bt BRANCH_LOCATION

Both cores include a convenient conditional execution (IFcc) instruction to tell the device to execute a line of code only if a specified condition is met. As with the change-of-flow instructions, the DSP56300 conditions are based on the eight possible condition codes in the status register while the SC140 conditional execution statement uses the T bit. The operation is similar between the two devices, but since the SC140 has multiple operating units it includes a mechanism for splitting an execution set into separate subgroups. Three IF instructions support this functionality: if-true (IFt), if-false (IFf), and if-always (IFa):

- Instructions grouped with the IFt execute if the T bit has a value of 1.
- Instructions grouped under IFf execute if the T bit has a value of 0.
- Instructions grouped with IFa execute unconditionally.

Example 13 shows a possible use of conditional execution in the SC140 core and how it contrasts with the DSP56300 core. The `mac`, `clr`, and `inc` instructions execute only if the T bit is set. The `lsl`, `neg`, and `dec` instructions execute regardless of the T bit setting.

Example 13. Conditional Execution Comparison

DSP56300 code:	SC140 code:
cmp x0,a	cmphi d0,d1
iflt mac x0,x1,a	[
	ift mac d0,d1,d1
	clr d5
	inc r1
	ifa lsl d4,d9
	neg d10
	dec r2
	]

2.3.2 Delay Slot

The SC140 core includes additional change-of-flow instructions with a “d” suffix that use the delay slot and can significantly improve code efficiency. Because of the pipeline activity when a change of flow occurs, an additional execution set may execute before the change of flow. This additional execution set is placed into the delay slot, in the additional line of code immediately after the change-of-flow instruction. In **Example 14**, the multiply-accumulate-with-rounding is completed prior to the start of execution at BRANCH_LOCATION. When code is converted from the DSP56300 core, it is helpful to remember these delay slot instructions because they can improve cycle time for your application.

Example 14. Delay Slot Usage

```
SC140 code:
cmphi d0,d1
jtd BRANCH_LOCATION
macr d0,d1,d1   macr d2,d3,d3   macr d4,d5,d5   macr d6,d7,d7
```

2.3.3 Bit Mask Instructions

In both the DSP56300 and SC140 cores, bit checking and manipulation is available to support software semaphore handling. The primary difference between the two cores is that the bit-test-change instructions of the SC140 core use a bit mask to define which bits can be tested and changed. Therefore, the SC140 core can test multiple bits in a single cycle. The DSP56300 instructions test a single bit. The DSP56300 core includes two instructions for this function: `bchg` (bit test and change) and `bclr` (bit test and clear). The SC140 can also test a bit directly in memory, but it uses a separate instruction mnemonic to indicate this variation. Both devices implement this testing as a read-modify-write operation that uses two memory accesses on the bus. Bit mask instructions on the SC140 core execute in the bit mask unit. This unit uses one of the AGU buses, so when these instructions are used they take one of the AGU slots in an execution set. **Table 2** lists the SC140 bit mask instructions. The SC140 bit-masked test instructions set the T bit, so a subsequent change-of-flow operation can be used to modify the program counter based on the status of a bit located in either a register or memory.

Table 2. SC140 Bit Mask Instructions

Instruction	Description
BMCHG	Bit-masked change a 16-bit operand
BMCHG.W	Bit-masked change a 16-bit operand in memory
BMCLR	Bit-masked clear a 16-bit operand
BMCLR.W	Bit-masked clear a 16-bit operand in memory
BMSET	Bit-masked set a 16-bit operand
BMSET.W	Bit-masked set a 16-bit operand in memory
BMTSET	Bit-masked test and set a 16-bit operand
BMTSET.W	Bit-masked test and set a 16-bit operand in memory
BMTSTC	Bit-masked test a 16-bit operand if clear
BMTSTC.W	Bit-masked test a 16-bit operand if clear in memory
BMTSTS	Bit-masked test a 16-bit operand if set
BMTSTS.W	Bit-masked test a 16-bit operand if set in memory

3 C Programming

Because C is a high-level language, conversion of C code between the DSP56300 and SC140 cores does not require as much concern with architectural intricacies. However, most DSP56300 C code has optimizations that make it non-ANSI-C-compliant. In addition, calling conventions between C and assembly routines in the two cores are not the same. One method for conversion is to convert the DSP56300 code to ANSI C code and then add device-specific optimization directives and assembly function calls as needed for an application's performance requirements. This section describes points to keep in mind when you are converting C code from the DSP56300 core to the SC140 core. In addition, the web site listed on the back cover of this document provides many application notes with C programming examples for the SC140 core. This section assumes that you are using the Tasking™ C compiler in combination with the Suite56 DSP tools for the DSP56300 and the Metrowerks CodeWarrior compiler for the SC140 core.

For the DSP56300 core, the Tasking compiler uses an explicit fractional type, `_fract`. In addition, type casting can be used in DSP56300 C code, such as `_CF(X) *(_fract *)&(X)`, to convert a variable to fractional. For integers, primitives are used to indicate integer arithmetic operations. The explicit type definitions, type casting, and primitives must be carefully coded to ensure that the data is handled properly. Keep in mind that the size of an integer in the SC140 is 32-bits, while an 'int' for the DSP56300 is 24-bits.

The SC140 core follows the ITU standards in defining intrinsics to handle fractional operations and allowing integer arithmetic to be ANSI C code. **Example 15** shows a comparison of how integer and fractional variables can be defined and then used in C. Below the C code is the assembly equivalent of the code.

Example 15. Integer and Fractional with the SC140 Compiler

```
long a;
short b,c;
a = a + b * c;
```

```
move.w (r0),d0
imac d0,d1,d2
```

```
#include "prototypes.h"
 long a;
 short b,c;
 a = L_mac(a,b,c);
```

```
move.f (r0),d0
mac d0,d1,d2
```

The C code is easy to read, and the compiler efficiently implements the code in SC140 assembly. The `prototypes.h` file includes the intrinsics used by the SC140 compiler. When you are working with standard ITU code, this file can replace the `basic_op.h` definition file. `Prototypes.h` also includes intrinsics that support double-precision arithmetic, generating assembly code for 32-bit × 32-bit operations while remaining at the C programming level. An option is included for translating a C statement directly into an assembly instruction such as `mpyus`. Additional intrinsics translate into multi-instruction operations for a full-precision result.

As noted earlier, the SC140 stack is a software stack that operates differently from the stack on the DSP56300 core. The stack location can be defined in the linker command file so that the compiler reserves that space for stack usage. The CodeWarrior start-up file uses this information to initialize the stack pointer. If this method is not used, the stack pointers must be explicitly initialized using inline assembly commands. **Example 16** shows in-line assembly for initializing both the exception and normal stack pointers to 0x30000 and 0x35000, respectively.

Example 16. Stack Initialization and Inline Assembly

```
asm(" move.l #$30000,r7 ");
asm(" move.l #$35000,r6 ");
asm(" tfra r6,osp ");              /* set ESP */
asm(" tfra r7,sp ");               /* set SP */
```

Notice that the format of inline assembly is somewhat modified from the format of the DSP56300 code. In DSP56300 code, an inline assembly line of code has the form `_asm("nop");`. The SC140 compiler allows a label to be defined in the inline assembly, so the first space after the `asm("` is for a label, if desired. Also, the _ symbol is not used with `asm` on the SC140 core. Tasking uses the following calling conventions for passing parameters, saving, and restoring registers:

- First parameter: passed in A if a numeric scalar, r0 if an address.
- Second parameter: passed in B if a numeric scalar, r4 if an address.
- Third parameter: passed in X0 if a numeric scalar, r1 if an address.
- Fourth parameter: passed in X1 if a numeric scalar, r5 if an address.
- Fifth parameter: passed in Y0 if a numeric scalar, r2 if an address.
- Sixth parameter: passed in Y1 if a numeric scalar.
- Subsequent parameters: passed on the stack.
- Return value: numeric value returned in A, address value returned in r0.

- Calling function must save registers, unless the save-by-callee calling convention is selected with the `_callee_save` keyword.

Note: If a 48-bit long data type is needed, the entire X or Y register is used.

The SC140 core uses different conventions. The *SC140 Application Binary Interface* document defines the rules for passing parameters, saving, and restoring registers with the SC140 core. Code that is converted from DSP56300 core to the SC140 core must modify how the calls are made and ensure that all registers are saved appropriately. Consider the C code in **Example 17**.

Example 17. C Function

```
main()
{
   ...
   sum = add_nums(var1, var2, var3, var4, var5);
   printf("sum = %d\n", sum);
   ...
}
```

Assuming that the `add_nums` function is implemented in assembly, you must be aware of which registers and memory locations are to hold the passed variables. Here are some rules to keep in mind from the SC140 ABI:

- First parameter: passed in d0 if a numeric scalar, r0 if an address.
- Second parameter: passed in d1 if numeric scalar, r1 if an address.
- Subsequent parameters: passed on the stack.
- Return value: numeric value returned in d0, numeric address value returned in r0.
- Called function *must* save and restore the following registers (if used): d6, d7, r6, r7.
- Calling function must save and restore any other registers before and after the function call.

For **Example 17**, `var1` is passed in d0. `var2` is passed in d1. `var3-var5` are passed on the stack. The result of the function is returned in d0. The variables passed on the stack are placed on the stack prior to the status register and return address for the change of flow. The assembly function can then use the stack for local variables and any additional saved registers. These must be removed from the stack prior to the return from subroutine. For **Example 17**, a function prototype for the assembly function must be defined, as shown in **Example 18**.

Example 18. SC140 Function Prototype

```
short add_nums{short var1, short var2, short var3, short var4, short var5);
```

For the DSP56300 core, this prototype is similar to **Example 19**. In contrast with the SC140, the Tasking C code must inform the compiler that this is an assembly function, use the `_fract` type, and specify the memory bank of the variable location. The SC140 compiler simplifies prototype definitions.

Example 19. DSP56300 Function Prototype

```
_asmfunc _fract add_nums{_fract _Y var1, _fract _Y var2,
                         _fract _Y var3, _fract _Y var4, _fract _Y var5);
```

The SC140 assembly language function provides the function marker using the label `_add_nums`. The `_` prefix indicates that the function can be called from C.

Example 20. SC140 Assembly Program

```
    section .text local
    global _add_nums

_add_nums
    push d6             push d7
    push r6             push r7
    add d1,d0,d0              ; add var1 and var2
    move.w (sp-10),d1         ; get var3 from stack
    add d1,d0,d0              ; add var3
    move.w (sp-12),d1         ; get var 4 from stack
    add d1,d0,d0              ; add var4
    move.w (sp-14),d1         ; get var5 from stack
    add d1,d0,d0              ; add var5
    pop r6              pop r7
    pop d6              pop d7

    rts                       ; sum is in d0
```

This example shows how interfacing between C and assembly is simpler in SC140 programming than in DSP56300 programming. Also, the standard intrinsics for handling fractional operations make it a minor task to run standard telecommunication algorithms on the SC140 core.

4 Software Tools Support

Suite56 tools for DSP56300 software development and CodeWarrior tools for SC140 software development both support DSP coding, but the CodeWarrior tools are a superset of the Suite56 tools. Similar to Suite56, CodeWarrior includes an assembler, linker, compiler, command-line debugger, and simulator. However, CodeWarrior also includes OS support, cache performance analysis, profiling, optimized code debugging, and more—all within a graphical user interface. CodeWarrior provides a seamless code development environment that you will find convenient.

The heart of the software development tools is the SC140 compiler, which performs so well in benchmarks that many DSP algorithms previously requiring hand-coding of critical sections in assembly can now remain completely in C. If you decide to hand-code some sections in assembly, you will find that the assembly macros are the same and that the assembly directives are very similar between the two cores. There are some changes due to the difference in addressing size (byte-addressing on the SC140), the SC140 unified memory addressing, and alignment directives for fetch sets. The linker and compiler of the SC140 core and DSP56300 core differ significantly. Be sure to review the linker and compiler manuals to become familiar with the available options. For a quick and easy start using the tools provided for the SC140 core, use any available ANSI C code and run it using the SC140 stationary provided with the CodeWarrior tools. After the ANSI C code is running, additional study of the C compiler options and linker directives will help you to improve code efficiency. Metrowerks provides extensive documentation detailing the features of the CodeWarrior environment.

5 Appendix

Table 3 shows the DSP56300 instructions paired with their similar SC140 instructions. The SC140 core does not always support an instruction feature of the DSP56300 core. Also, the SC140 core supports instructions not available in the DSP56300 core. The instruction cache control instructions are not included in the table. Use of an instruction cache with the SC140 core is device-specific and controlled through cache implementation.

Table 3. Instruction Set Comparison

DSP56300	SC140	Description
Arithmetic Instructions		
ABS	ABS	Absolute value
ADC	ADC	Add long with carry
ADD	ADD	Add
—	ADD2	Add two words
ADDL	—	Shift left and add
—	ADDNC.w	Add without changing carry bit in status register
ADDR	—	Shift right and add
—	ADR	Add and round
ASL	ASL	Arithmetic shift left
ASL	ASLL	Multi-bit arithmetic shift left
—	ASLW	Word arithmetic shift left (16-bit shift)
ASR	ASR	Arithmetic shift left
ASR	ASRR	Multi-bit arithmetic shift right
—	ASRW	Word arithmetic shift right (16-bit shift)
CLR	CLR	Clear accumulator
CMP/CMPM/CMPU	CMPEQ/CMPGT/CMPHI	Compare instructions
DEC	DECEQ/DECGE	Decrement by one
DIV	DIV	Divide iteration
DMAC	DMACSS/DMACSU	Double-precision multiply-accumulate with right shift
—	IADD	Add integers
—	IMAC	Multiply-accumulate integers
—	IMACLHUU	Multiply-accumulate unsigned integers
—	IMACUS	Multiply-accumulate unsigned integer and signed integer
—	IMPY.w	Multiply integer
—	IMPYHLUU	Multiply unsigned integer and unsigned integer
—	IMPYSU/IMPYUU	Multiply mixed signed/unsigned integers
INC	INC	Increment by one (integer)
—	INC.f	Increment by one (fractional)
MAC	MAC	Signed multiply-accumulate

Table 3. Instruction Set Comparison (Continued)

DSP56300	SC140	Description
MAC	MACSU/MACUS/MACUU	Multiply-accumulate with mixed signed/unsigned
MACI	MAC	Signed multiply-accumulate with immediate operand
MACR	MACR	Signed multiply-accumulate with round
MACRI	MACR	Signed multiply-accumulate and round with immediate
MAX	MAX	Transfer maximum signed value
—	MAX2	Transfer two 16-bit maximum signed values
—	MAX2VIT	Special MAX2 version for Viterbi kernel
MAXM	MAXM	Transfer maximum magnitude value
—	MIN	Transfer minimum signed value
MPY	MPY	Signed multiply
MPY	MPYSU/MPYUS/MPYUU	Multiply with mixed signed/unsigned
MPYI	—	Multiply with immediate operand
MPYR	MPYR	Multiply signed fractions and round
MPYRI	—	Multiply signed fractions and round with immediate
NEG	NEG	Negate accumulator
NORM	—	Norm accumulator iteration
NORMF	—	Fast accumulator normalization
RND	RND	Round accumulator
—	SAT.F	Saturate fraction value
—	SAT.L	Saturate long value
SBC	SBC	Subtract long with carry
—	SBR	Subtract and round
SUB	SUB	Subtract
—	SUB2	Subtract two words
SUBL	SUBL	Shift left and subtract
—	SUBNC.w	Subtract without changing carry bit in status register
SUBR	—	Shift right and subtract
Tcc	TFRT/TFRF	Transfer conditionally
TFR	TFR	Transfer data ALU register
TST	TSTEQ/TSTGE/TSTGT	Test accumulator
Logical Instructions		
AND	AND	Logical AND
ANDI	AND	Logical AND
CLB	CLB	Count leading bits
EOR	EOR	Logical exclusive OR
EXTRACT	EXTRACT	Extract bit field

Table 3. Instruction Set Comparison (Continued)

DSP56300	SC140	Description
EXTRACTU	EXTRACTU	Extract unsigned bit field
INSERT	INSERT	Insert bit field
LSL	LSLL	Logical shift left
LSR	LSR	Logical shift right by one bit
LSR	LSRR	Multi-bit logical shift right by
—	LSRW	Word logical shift right (16-bit shift)
MERGE	—	Merge two half words
NOT	NOT	One's complement
OR	OR	Logical OR
ORI	—	OR immediate with control register
ROL	ROL	Rotate one bit left through the carry bit
ROR	ROR	Rotate one bit right through the carry bit
—	SXT.b, SXT.l, SXT.w	Sign extend byte/long/word
—	ZXT.b, ZXT.l, ZXT.w	Zero extend byte/long/word
Bit Manipulation Instructions		
BCHG	BMCHG	Bit test and change/Bit-mask change
BCLR	BMCLR	Bit test and clear/Bit-mask clear
BSET	BMTSET	Bit test and set/Bit-mask test and set
—	BMSET	Bit-mask set a 16-bit operand
BTST	BTSTS/BTSTC	Bit test if set, if clear
Loop Instructions See Section 2.2, *Hardware Loops,* on page 11		
Move Instructions See Section 2.3, *Instruction Set Differences,* on page 14		
Change-of-Flow Instructions		
Bcc	BT/BF, BTD/BFD	Branch conditionally, see **Section 2.3.1** for details
BRA	BRA, BRAD	Branch always
BRCLR	—	Branch if bit clear
BRSET	—	Branch if bit set
BScc	—	Branch to subroutine conditionally
BSCLR	—	Branch to subroutine if bit clear
BSR	BSR, BSRD	Branch to subroutine
BSSET	—	Branch to subroutine if bit set
Jcc	JT/JF, JTD/JFD	Jump conditionally
JCLR	—	Jump if bit clear
JMP	JMP, JMPD	Jump
JScc	—	Jump to subroutine conditionally
JSCLR	—	Jump to subroutine if bit clear
JSET	—	Jump if bit set
JSR	JSR, JSRD	Jump to subroutine

Table 3. Instruction Set Comparison (Continued)

DSP56300	SC140	Description
JSSET	—	Jump to subroutine if bit set
RTI	RTE, RTED	Return from interrupt/exception
RTS	RTS, RTSD	Return from subroutine
—	RTSTK, RTSTKD	Force restore PC from the stack, updating SP
Program Control Instructions		
DEBUG	DEBUG	Enter debug mode
DEBUGcc	—	Enter debug mode conditionally
—	DEBUGEV	Signal debug event
—	DI	Disable interrupts
—	EI	Enable interrupts
IFcc	—	Execute conditionally without CCR update
IFcc.U	IFA,IFF,IFT	Execute conditionally, see **Section 2.3.1** for details
ILLEGAL	ILLEGAL	Trigger an imprecise illegal instruction exception
—	MARK	Push the PC into the trace buffer
NOP	NOP	No operation
REP	—	Repeat next instruction
RESET	—	Reset on-chip peripheral devices
STOP	STOP	Stop processing
TRAP	TRAP	Execute a precise software exception
TRAPcc	—	Conditional software interrupt
WAIT	WAIT	Wait for interrupt

Table 4 lists the AGU arithmetic instructions that are new on the SC140 core.

Table 4. SC140 AGU Arithmetic Instructions

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

Table 4. SC140 AGU Arithmetic Instructions (Continued)

Instruction	Description
LSRA	Logical shift right (32-bit)
SUBA	Subtract
SXTA.B	Sign extend byte
SXTA.W	Sign extend word
TFRA	Register transfer
TSTEQA	Test for equal
TSTEQA.w	Test for equal on lower 16 bits
TSTGEA	Test for greater than or equal
TSTGTA	Test for greater than
ZXTA.b	Zero extend byte
ZXTA.w	Zero extend word

NOTES:

NOTES:

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 München, 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

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

AN2715
Rev. 1
11/2004

freescale™
semiconductor