Chapter 5
Program Control

This chapter describes the SC140 program control features, including pipeline, hardware loop execution, stack support, processing states and exception processing.

The SC140 core, being a multiple ALU machine, has special hardware logic to support the need to issue up to six instructions to six different execution units at the same time. When two or more instructions are being issued to two or more execution units in the same clock cycle, these instructions are labelled as grouped. The assembly programmer or the C compiler can specify in the DSP code which instructions are grouped together, according to the SC140 programming rules. When the assembler compiles the DSP code, it specifies in the encoding whether an instruction stands alone, or whether it is grouped with other instructions. In each clock cycle, the dispatch logic determines how many instructions are grouped. Each block of instructions issued to the execution units at a certain clock cycle is called an execution set. Each block of eight words read from the program memory, associated with a certain address, is called a fetch set.

5.1 Pipeline

This section describes how instructions are processed in the SC140 core pipeline.

The SC140 pipeline consists of five stages:

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

The first three stages are implemented in the program sequencer unit (PSEQ) and the last two stages are implemented in the AGU and in the Data ALU.

The multiple execution units of the SC140 core enable it to execute several instructions in each cycle. The core contains four ALUs and two AAUs, hence up to six instructions can be executed in a single cycle, with a maximum of eight words in total.

To support the need for parallel execution, the core uses a variable length execution set (VLES) architecture with a static-grouping mechanism. This method enables several instructions to be grouped together to form an execution set, which is dispatched to the execution units.

This section describes the SC140 pipeline stages. For a detailed description of SC140 core parallel memory accesses, see Section 6.4, “Parallel Memory Accesses,” on page 6-14.

5.1.1 Instruction Pipeline Stages

Figure 5-1 illustrates the five instruction pipeline stages.

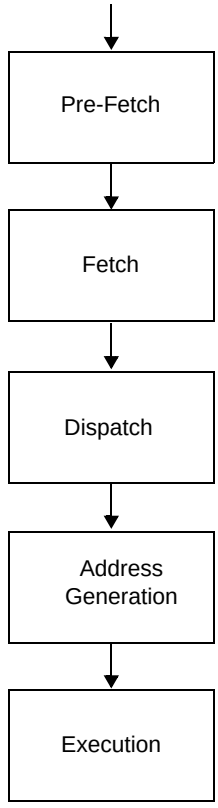


Figure 5-1. Instruction Pipeline Stages

Table 5-1 provides an overview of the operations performed at each stage of the pipeline.

Table 5-1. Pipeline Stages Overview

Pipeline Stage	Description
Pre-Fetch	• Generate addresses for program fetch • Increment fetch counter (FC)
Fetch	• Fetch set, consisting of eight instruction words, read from memory
Dispatch	• Dispatch instructions • Decode AGU instructions
Address Generation	• Decode Data ALU instructions • Generate addresses for data load or store operations • Perform address calculation • Perform AGU arithmetic instructions • Update AGU registers with calculation results
Execution	• Read source operands to Data ALU • Read source register for memory store operations • Multiply and add • Write Data ALU results to destination registers • Write destination register for memory load operations

5.1.1.1 Instruction Pre-Fetch and Fetch

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

In the pre-fetch stage, the address of the fetch set is driven on the program address bus (PAB) along with the read strobe, to enable the read process in the memory logic. While the address is issued to the memory, the fetch counter (FC) in the PSEQ is recalculated for the next program memory read. These operations occur in parallel. The address can be generated by the PSEQ:

- For the normal program flow and the exception program flow, or
- For change-of-flow instructions in the AGU

The fetch stage, which follows the pre-fetch stage, is dedicated to waiting for the memory access to be completed. Memory access is completed when the program memory writes the value on the 128-bit wide program data bus (PDB), enabling the PSEQ to sample it. The pipeline process then proceeds to the next stage.

5.1.1.2 Instruction Dispatch

After the fetch set is read from the memory to the PSEQ, the dispatch stage starts. During this stage, the PSEQ determines the nature of the instructions (Data ALU type or AGU type), and also which instructions, if any, should be executed in parallel with others, meaning, which instructions are grouped together. The cycle length of a particular execution set is determined by the longest instruction (in terms of timing) included in that execution set. This stage includes the actual decoding of all AGU instructions.

5.1.1.3 Address Generation

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

5.1.1.4 Execution

At the SC140 execution stage, all Data ALU arithmetic calculations are performed and data is read from its source and written to its destination, either register or memory. In the Data ALU, this stage includes the following:

- Reading the data operands
- Arithmetic operation
- Writing the results to the destination register

5.1.2 PSEQ Architecture

The PSEQ implements three of the five stages in the SC140 pipeline. This includes the two instruction fetch stages and the instruction dispatch stage. In addition, the PSEQ is responsible for controlling the machine in the following cases:

- Hardware loops
- Change-of-flow instructions
- Exception processing

The PSEQ is also responsible for the processing state of the SC140 core. The processing state options are:

- Normal
- Exception
- Debug
- Reset
- Wait
- Stop

Refer to Section 5.4, “Processing States,” for further information.

The PSEQ consists of the following three hardware blocks:

- Program control unit (PCU)—Controls the machine pipeline in all core processing states, including instances of change-of-flow instructions, exception processing and low power modes.
- Program dispatch unit (PDU)—Performs the preliminary decoding of the current instructions being dispatched to the different execution units.
- Program address generator (PAG)—Calculates the program address and controls the hardware loops and the hardware.

5.1.3 Instruction Issuing and Grouping Mechanism

The SC140 core contains two AAUs and four ALUs, enabling six instructions to be executed in parallel, four instructions to ALUs and two instructions to the AAUs. This section describes the instruction issue and execution paradigm for the core.

5.1.3.1 Prefix Words

The core architecture uses dedicated instructions, called prefix instructions, to provide architecture enhancements in various areas. The prefix instruction is a part of the execution set, but it is not issued directly to any of the execution units. The PSEQ uses the prefix instructions in the following circumstances:

- To enhance the grouping options
- To expand the accessible register space
- To implement hardware loops using the loop marks
- For the conditional execution instruction (IFc)

Since the fetch set is eight words long and the maximum number of execution units required for executing a full execution set is six (four Data ALU instructions and two AGU instructions), the two prefix words do not usually affect performance.

5.1.3.2 Grouping Mechanism

The SC140 grouping mechanism includes two methods for conveying grouping information:

- The serial, or non-prefix, grouping method, which uses the two most significant bits in the instruction.
- The prefix grouping method, which uses a one-word or two-word prefix for an execution set.

The core determines which instructions should be issued to the execution units in a certain clock cycle according to the encoding.

In serial grouping, the value 00 in the two MS bits of an instruction word indicates that this word is to be grouped with the next instruction word. An instruction with a value other than 00 in its two MS bits is considered the last instruction in the set, and marks the execution set boundary.

If a prefix exists at the beginning of an execution set, the core uses it to determine the grouping information, including the number of instruction words grouped in the execution set.

Figure 5-2 illustrates the serial and prefix methods of the SC140 grouping mechanism.

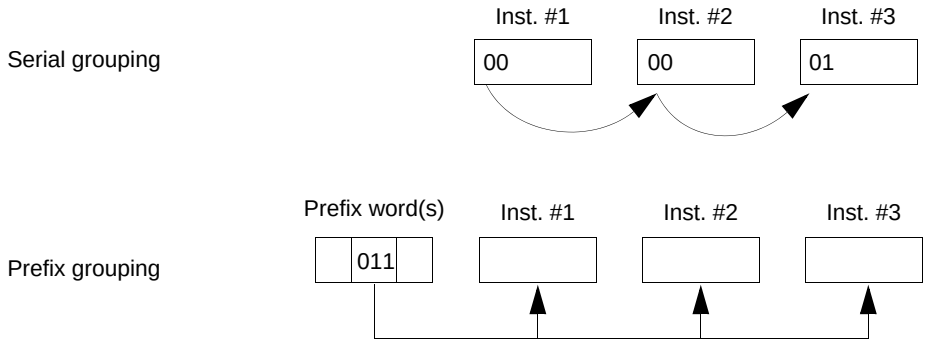


Figure 5-2. Instruction Grouping Methods

Refer to Section 6.2, “Instruction Grouping,” on page 6-2, for further information about the grouping mechanism.

5.1.4 Timing of Major Instructions

While all the instructions that are needed in the DSP part of the code take one cycle to execute and can be grouped together and executed simultaneously, several other instructions, mostly used in the control part of the code, take more than one cycle to execute. Some of these are change-of-flow instructions, some use a special addressing mode, and some operate on memory and require additional clock cycles.

The timing of an execution set is determined by the longest instruction (in terms of timing) in the set.

Refer to Section 6.3, “Timing,” on page 6-10, for more details about instruction timing.

5.1.4.1 Data ALU Instructions

All the Data ALU instructions, which are the most timing-critical instructions in the DSP kernels, take only one cycle to execute. These include the following instructions:

- Multiply-accumulate (MAC)
- Multiply (MPY)
- ADD
- SUB
- Compare
- Shift
- Test

5.1.4.2 Data Moves

Most of the data move instructions take one cycle to execute. This includes the byte, word, two-word, long-word, four-word and two long-word, signed or unsigned, both from memory to register and from register to memory, in most of the addressing modes, with the exception of the pre-calculation addressing modes. The addressing modes (Rn + N0), (Rn + Rm), (Rn + x), (Rn + xxxx), (SP - xx) and (SP + xxxx) require one additional clock cycle to calculate the address of the memory access.

5.1.4.3 Change-of-Flow Timing

The timing of the change-of-flow instructions is usually most affected by the access time to memory and by the number of stages in the architecture pipeline. The SC140 core implements a five-stage pipeline, with two stages dedicated to memory access. This results in a penalty of two clock cycles for unconditional change-of-flow instructions which use immediate values, and in a penalty of three clock cycles for the PC-relative change-of-flow instructions. Conditional change-of-flow instructions, where the condition is found to be true, meaning the change-of-flow operation is taken, always has a penalty of three cycles. When a conditional change-of-flow is determined as not taken, meaning the condition is found to be false, there is no timing penalty.

The core implements a mechanism for fast call-return from subroutine. This mechanism includes a single level cache register (RAS) to keep the last return address from a subroutine.

Refer to Section 5.3.4, “Fast Call-Return from Subroutines,” for a more detailed description of the fast call-return mechanism.

5.1.4.4 Delayed Change-of-Flow Timing

When a change-of-flow instruction is executed, the core must wait for the destination address to be fetched from memory. The cycles that are lost while waiting for the destination address are referred to as delay slots. Since it is possible to use the delay slots of the change-of-flow operation to continue the execution of the previously fetched instructions, special delayed instructions are added to the instruction set. These instructions use part or all of the delay cycles to execute one additional execution set, thus reducing the penalty for the change-of-flow operation. Refer to Section 6.3.2, “Change-of-Flow Instruction Timing,” on page 6-10, for further details.

5.1.4.5 Bit Mask Instructions

The SC140 core includes various instructions which allow bit mask operations. These instructions are helpful when the content of several bits needs to be changed or tested at the same time. The bit mask instructions include the following:

- Bit mask set
- Bit mask clear
- Bit mask change
- Bit mask test
- Bit mask test and set

The bit mask instructions are of the read-modify-write type, meaning they include three steps:

1. Read the operand
2. Change (set, clear, change)
3. Write back to the source, which is also a destination

This type of instruction takes two clock cycles to execute for the simple addressing modes, and three clocks for the addressing modes that require pre-calculation of the address.

Refer to Appendix A, “SC140 DSP Core Instruction Encoding,” for a full description of the bit mask instructions.

5.1.4.6 Instruction Categories Timing Summary

Table 5-2 summarizes the timing of the various categories of SC140 basic instructions.

Table 5-2. Basic Instruction Categories Timing Summary

Basic Instruction Category	Example/Condition	Number of Clock Cycles
Data ALU	MAC D0, D1, D2	1
Move with simple addressing	MOVE.W (R0) +N2, D3	1
Move with address pre-calculation	MOVE.W (R5 + N0), D4	2
Simple change-of-flow	JMP _dest	3
PC-relative change-of-flow	BRA _dest	4
Conditional change-of-flow	If condition is true	4
	If condition is false	1
Delayed change-of-flow	Simple	3 minus cycles used by execution set in delay slot
	PC-relative	4 minus cycles used by execution set in delay slot
BMU	BMSET #$1010, (R0)	2

5.1.5 Change-of-Flow Instructions

The change-of-flow instructions include branches, jumps, conditional branches, conditional jumps, and other instructions that affect the program counter and software stack. Program control instructions may affect or be affected by status register bits as specified in the instruction. In the SC140 instruction set naming convention, “jump” signifies instructions using explicit destination address (either absolute or in a pointer), while “branch” signifies instructions that use a PC-relative offset to specify the destination address.

Jumps and branches to subroutines (JSR/BSR) include implicit stack push operations to the stack. Similarly, return from subroutines or exceptions (RTS/RTSTK/RTE) include implicit pop operations from the stack.

Change-of-flow instructions usually take longer to execute because the pipeline is flushed during their execution. In order to use this time more effectively, most of the change-of-flow instructions have a delayed version that enables the execution of one execution set while the pipeline is filling up again. The delayed instruction effectively executes one or more cycles less than the non-delayed version.

In Example 5-1, the MOVE.W instruction is logically executed before the jump. The delayed change-of-flow instruction JMPD and the execution set are a non-interruptible sequence.

Example 5-1. Change-of-Flow Instruction

```
JMPD        destination_label
MOVE.W      (R0+N0),D0
```

Table 5-3 lists the change-of-flow instructions.

Table 5-3. Change-of-Flow Instructions

Instruction	Description
BF	Branch if false
BFD	Branch if false (delayed)
BRA	Branch
BRAD	Branch (delayed)
BSR	Branch to subroutine
BSRD	Branch to subroutine (delayed)
BT	Branch if true
BTD	Branch if true (delayed)
JF	Jump if false
JFD	Jump if false (delayed)
JMP	Jump
JMPD	Jump (delayed)
JSR	Jump to subroutine
JSRD	Jump to subroutine (delayed)
JT	Jump if true
JTD	Jump if true (delayed)
RTE	Return from exception
RTED	Return from exception (delayed)
RTS	Return from subroutine
RTSD	Return from subroutine (delayed)
RTSTK	Force restore PC from the stack, updating SP
RTSTKD	Force restore PC from the stack, updating SP (delayed)

5.1.6 Program Control Instructions

The program control instructions include miscellaneous instructions for special control operations, such as:

- Conditional execution of an execution set
- Entry into low-power modes
- Operations for debug support
- TRAP and ILLEGAL, which force the chip into exception processing mode.

Certain instructions are individually controlled by the state of the T-bit in SR, for example, JT (Jump if True) is executed only it the T-bit is set. In addition, the following instructions enable control of the execution of the set as a whole, or sub-groups of the set:

- IFT (If True)
- IFF (If False)
- IFA (If Always) instructions.

For example:

```
IFT   ADD D0,D1,D2   MOVE.L (R0)+,D0
```

The set as a whole, including the ADD and MOVE instructions, is executed only if the T-bit is set. For better control, it is also possible to split the instructions in the execution set into two subsets, and conditionally control the activation of each subset independently. The following possibilities are supported:

```
IFT   subset1  IFA   subset2        ;execute subset 1 if T is set,
                                    ;execute subset 2 unconditionally

IFF   subset1  IFA   subset2        ;execute subset 1 if T is clear,
                                    ;execute subset 2 unconditionally

IFT   subset1  IFF   subset2        ;if T is set, execute subset 1,
                                    ;else execute subset 2.
```

The instructions in the subsets may themselves be conditional, for example, TFRT, JF, which may further add to the selectivity control.

A number of restrictions apply to the number and length of instructions that can appear in the subsets. Refer to Section 6.2.1.1, “General Grouping Restrictions,” on page 6-4, for details.

The instructions DEBUG, DEBUGEV and MARK are intended to provide software debug support and are relevant when debugging using the EOnCE block. Refer to Chapter 4, “Emulation and Debug (EOnCE),” for further details.

Table 5-4 lists the program control instructions.

Table 5-4. Program Control Instructions

Instruction	Description
NOP	No operation, not dispatched to an execution unit
IFA	Execute current execution set or subset unconditionally
IFF	Execute current execution set or subset if the T-bit is clear
IFT	Execute current execution set or subset if the T-bit is set
DI	Disable interrupts (sets the DI bit in SR)
EI	Enable interrupts (clears the DI bit in SR)
WAIT	Wait for interrupt (low power stand-by)
STOP	Stop processing (lowest power stand-by)
TRAP	Execute a software exception
ILLEGAL	Trigger an illegal instruction exception
DEBUG	Enter Debug mode
DEBUGEV	Signal debug event
MARK	Push the PC into the trace buffer

5.2 Hardware Loop Execution

One of the most important features of a DSP engine is an efficient loop execution machine. The SC140 core has four fully optimized loop machines, which enable loop execution with up to four levels of loop nesting. The loop machine programming model is part of the PSEQ programming model and includes four pairs of registers, specifying the start address of the loop and the number of times the loop is to be repeated.

5.2.1 Hardware Loop Programming Model

There are four pairs of registers, with two registers in each pair:

- Loop start address registers (SA0, SA1, SA2, SA3)
- Loop counter registers (LC0, LC1, LC2, LC3)

Each pair is responsible for a single hardware loop. The functionality of each register pair is described in the sections that follow.

Figure 5-3 shows the programming model of the hardware loop model.

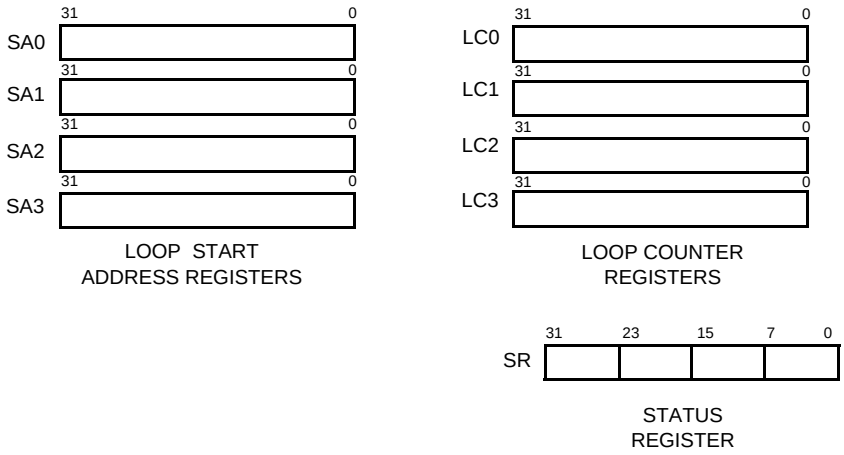


Figure 5-3. Hardware Loop Programming Model

This programming model holds the full loop state and can be saved and restored in interrupts and context switches.

5.2.1.1 Loop Start Address Registers (SAn)

The SAn registers are 32-bit read/write registers that are used to define the address of the first execution set of the loop.

5.2.1.2 Loop Counter Registers (LCn)

The LCn registers are 32-bit read/write registers used to define the number of times each loop is to be repeated. LCn always holds a 32-bit signed value. This means that the largest number of loop iterations is 2^{31}-1. Whenever a SKIPLS instruction is executed, the LCn is checked. If its value is less than or equal to zero, the program jumps to the address specified by the SKIPLS instruction. This is typically, but not necessarily, the end of the loop.

If there is no SKIPLS instruction before entering the loop and the LCn is less than or equal to one, the loop ends after one iteration.

Upon reaching the decision point for iteration (two execution sets before the end of the loop for long loops, or the first set of the loop for short loops), the LCn is compared to one, to detect loop termination. If the value of LCn is less than or equal to one, the loop terminates and the LCn register is cleared. If the value of LCn is greater than one, LCn is decremented by one and the loop proceeds to its next iteration.

5.2.1.3 SR Loop Flag Bits

Certain status bits in the SR are associated with hardware loop initiation and execution. These bits are set and cleared by special loop instructions such as DOENn and various loop conditions, or by an explicit write to the SR. The bits are:

- Loop flag bits—Four loop flag bits (LF0, LF1, LF2, LF3) are defined in the SR, one for each hardware loop. The bit is set when the loop is initiated by either the DOENn or the DOENSHn instruction, and cleared when the loop terminates.
- Short loop flag bit—This bit (SLF) is set when the loop is initiated by the DOENSHn instruction, and is cleared when the loop terminates.

5.2.2 Assembly Syntax for Hardware Loops

The loop definitions are as follows:

- Loop body—The execution sets that are iterated during loop execution.
- Long loop—A loop body that consists of three or more execution sets.
- Short loop—A loop body that consists of one or two execution sets.
- SA—Start address. The address of the first execution set in a loop body. Do not confuse this with SA0, SA1, SA2 and SA3 which are register names, used to hold SA values.
- LA—Last address. The address of the last execution set in a loop body. In the case of a loop with only one execution set, the SA contains the last address.
- SA+1—Address of the execution set following SA. Similarly SA+2, and so on.
- LA-2—Address of the execution set that comes two execution sets before the execution set at LA. Similarly LA-1, and so on.
- LPMARKA and LPMARKB are two marker bits in the prefix instruction that identify different looping conditions. The LPMARK bits are typically not written by the user but are set automatically by the assembler.

Table 5-5 illustrates the location of these markers and their functionality in both short and long loops.

Table 5-5. LPMARKA and LPMARKB in Short and Long Loops

Loop Type	LPMARKA		LPMARKB	
	Location	Functionality	Location	Functionality
Short loop	SA	Identifies a single-execution set loop. Does not carry a timing overhead.	SA	Identifies a two-execution set loop. Does not carry a timing overhead.
Long loop	LA	Identifies a jump to SA if the loop is to iterate. Carries a timing overhead.	LA-2	Identifies a jump to SA if the loop is to iterate. Does not carry a timing overhead.

Refer to Appendix A, “SC140 DSP Core Instruction Encoding,” for further details.

5.2.3 Hardware Loop Initiation and Execution

The following steps are required to initiate a hardware loop:

1. A DOSETUPn instruction should be executed at some stage before the loop starts, except in the case of a short loop. This instruction writes the start address of that loop to the corresponding SAn register.
2. A DOENn or DOENSHn instruction should be executed to load the corresponding LCn register with the number of iterations for the loop. The corresponding loop flag bit is implicitly set when LCn is loaded with the loop iteration value. The SLF is set if the loop is initialized by a DOENSHn instruction.
3. Whenever a SKIPLS instruction is executed before entering the loop, the value of LCn is checked. If the value of LCn is less than or equal to zero, then the loop is skipped and the PC is loaded with the address specified in the SKIPLS instruction. If it is guaranteed that LCn is greater than zero, for example, if the loop is initialized by an immediate value, the SKIPLS instruction can be omitted.

The SAn is loaded by the DOSETUPn instruction. After the LCn is loaded and the LFn bit is set with the DOENn or DOENSHn instruction, the loop machine is ready for operation. In long loops, whenever the program reaches the execution set which appears two execution sets before the last execution set of the loop (LA-2), LCn is compared to one to detect loop termination. In short loops, the iterated execution sets are stored in internal buffers and iterated the appropriate number of times, according to the value stored in LCn. If the value of LCn is greater than one, the program address bus is loaded with the value stored in SAn, the LCn is decremented by one, and the loop is repeated. When the loop terminates, execution of the program continues normally, and the loop flag bit is cleared.

5.2.4 Loop Nesting

The core has four loop machines (LOOP0, LOOP1, LOOP2 and LOOP3), allowing loop execution with up to four levels of loop nesting.

A loop can be nested only within a loop that has a lower index. A loop is enabled when its corresponding LFn is set. The active loop is the enabled loop with the highest index. Only one loop can be active at a time.

Figure 5-4 shows an example of the loop nesting structure.

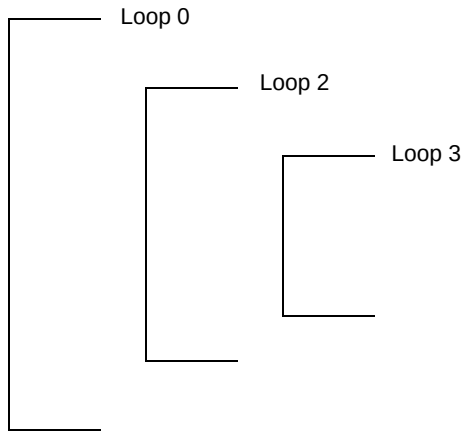


Figure 5-4. Loop Nesting

In Figure 5-4, the three loops, Loop 0, Loop 2 and Loop3 are enabled. Loop 3 is active until it has finished iterating, at which time Loop 2 becomes active. When Loop 2 stops iterating, Loop 0 becomes the active loop.

5.2.5 Iteration and Termination of Loops

The CONT instruction causes the current loop iteration to terminate before reaching the last execution set of the loop. If the value of LCn is greater than one, then the CONT instruction causes the program to branch to the address stored in SAn, the LCn is decremented by one and the loop is repeated. If the value of the LCn is less than or equal to one, then the CONT instruction causes the program to branch to the address specified by the CONT instruction. The LCn is cleared and the loop terminates (LFx is cleared).

The BREAK instruction causes the loop to terminate. The program address bus is loaded with the address specified by the BREAK instruction and the loop terminates (LFx is cleared), regardless of the value of LCn. The value of LCn is not changed.

5.2.6 Loop Instructions

Table 5-6 lists the loop instructions.

Table 5-6. Loop Instructions

Instruction	Operation
DOSETUPn <label>	Initialize register SAn with <label> address. Used only in long loops.
DOENn (reg or imm)	Activate loop n as a long loop.Performs the following operations: • Initializes LCn • Sets LFn in the SR
DOENSHn (reg or imm)	Activate loop n as a short loop.Performs the following operations: • Initializes LCn • Sets LFn and sets SLF in the SR
SKIPLS <label>	If LCn <= 0, jumps to <label>, clearing LFn.
CONT <label>	Within a loop, if the active LC > 1, jumps to SA and decrements LC. If LC <= 1 jumps to <label> and clears the LC register and the active LF.
CONTD <label>	A delayed version of the CONT instruction.
BREAK <label>	Within a loop, jumps to <label> and clears the active LF.

The instructions that activate the loop are either DOENn or DOENSHn. In nested loops, DOENn or DOENSHn must be re-executed in order to re-activate the inner loop. DOSETUPn is used to initialize SAn in long loops only and need not be re-executed if the value of the SAn register is unchanged. In short loop initialization, this instruction is not needed at all. SKIPLS, CONT, BREAK and their variants are optional, to be used only if needed.

The above instructions are assembled to mnemonics in the conventional way and disassembled normally. In addition, looping information is given to the machine with dedicated bits in the prefix. When coding a hardware loop in assembly, two loop-related assembly directives must be used:

- LOOPSTARTn—Marks SA, placed immediately before it.
- LOOPENDn—Marks LA, placed immediately after it.

By definition, a loop body is enveloped by LOOPSTART and LOOPEND directives.

The following is an example of a long loop:

Example 5-2. Long Loop

```
                dosetup0  _start0
                doen0     #$10
                move.w    (r3)+,d1
                skipls    _end0
                loopstart0
SA ──► _start0
                mac       d0,d1,d2   move.w   (r0)+,d0
                add       d5,d6,d4   move.w   (r1)+,d5
                sub       d3,d2,d4   inc d5
                mac       d0,d1,d6   move.w (r0)+,d7
LA ──►          sub       d5,d4,d4   inc d7
                loopend0
        _end0
```

Loop body

LPMARKB

The following is an example of a short loop of two execution sets:

Example 5-3. Short Loop, Two Execution Sets

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

LPMARKB

Loop body

The following is an example of a short loop of one execution set:

Example 5-4. Short Loop, One Execution Set

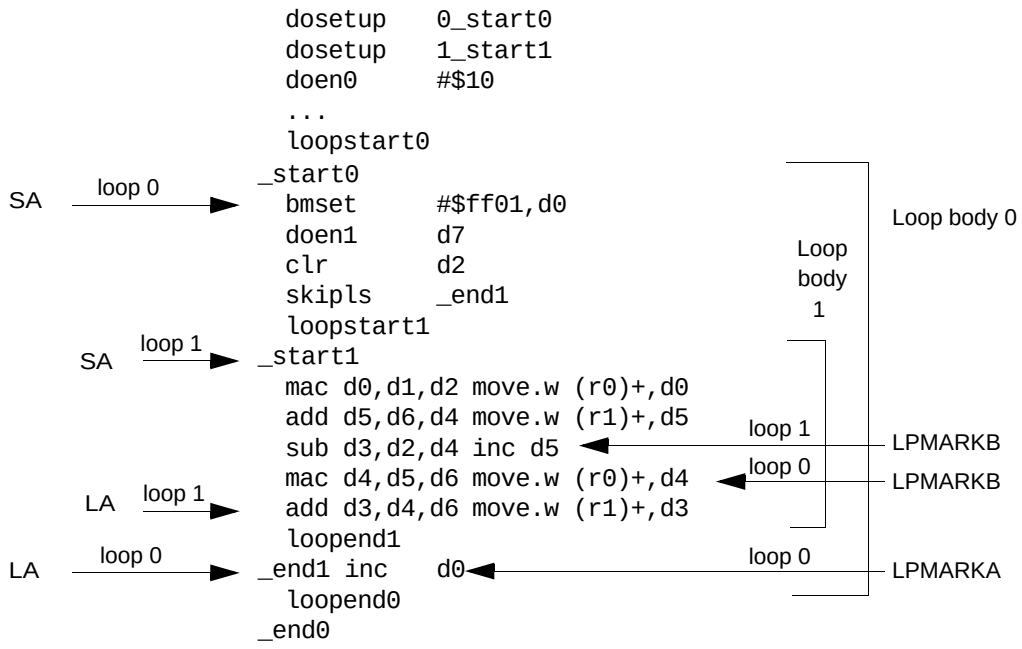
```
                doensh0   #$10
                ...
                loopstart0
SA, LA ──►      mac       d0,d1,d2   move.w (r0)+,d0
                loopend0
```

LPMARKA

Loop body

The following is an example of a nested loop:

Example 5-5. Nested Loop



In Example 5-5, the LPMARKA of loop 0 is set because of the SKIPLS instruction that can cause a skip over the LPMARKB of loop 0.

The assembler sets the appropriate prefix bits, adding a prefix word if necessary, with the loopstart or loopend information. In disassembly, these prefix bits (if used) appear, preceding the normal disassembled mnemonics of the set.

Example 5-6 shows the disassembly of the long loop in Example 5-2:

Example 5-6. Long Loop Disassembly

```
p:$00000000 2803 800c         = dosetup0 *+$c
p:$00000004 9050              = doen0 #<$10
p:$00000006 511b              = move.w (r3)+,d1
p:$00000008 2103 801a         = skipls >*+$1a
p:$0000000c 2111 5018         = mac d0,d1,d2 (r0)+,d0
p:$00000010 2e5a 5519         = add d5,d6,d4 (r1)+,d5
p:$00000014 94d0 6e3d 66ef    = lpmarkb sub d3,d2,d4 inc d5
p:$0000001a 2311 5718         = mac d0,d1,d6 (r0)+,d7
p:$0000001e 2e35 67ef         = sub d5,d4,d4 inc d7
```

5.2.6.1 Loop Timing

If the loop is not aligned, meaning that the first execution set is spread over two fetch sets, one stall cycle is added to the loop execution on each iteration of the loop. In every other case, no stall cycles are added to the loop execution time.

At the end of a long loop, if LCn is greater than one, encountering LPMARKB does not cause a timing penalty, while encountering LPMARKA results in a change-of-flow timing penalty.

5.2.7 Looping Restrictions

Looping operations are subject to a number of restrictions, including the following main areas of sensitivity:

- Loop nesting
- Instructions at the end of loops
- Minimum distance between looping instructions
- Change of flow and looping

Refer to Section 6.5, “Restrictions,” on page 6-16, for a more detailed description of all the restrictions.

5.3 Stack Support

This section covers the SC140 support for efficient multitasking. Multitasking creates the impression that the DSP is executing several tasks concurrently, when in reality it is only executing a single task at any given time. The SC140 core has many features that help software designers when implementing a software stack, and, more generally, when supporting a multitasking OS. These features include:

- Two stack pointers, one for the normal stack (NSP) and one for the exception stack (ESP), of which only one is active at a time (SP)
- Separate normal and exception modes
- Push and pop instructions
- Stack-oriented addressing modes

5.3.1 Normal and Exception Modes

One feature for efficient multitasking is the separation of normal and exception states. This allows separation between tasks and the OS. In this context, the OS includes interrupt service routines (ISR), since they are part of the interface to the physical hardware that the OS abstracts. The separation of normal and exception states allows the task memory requirements to be separated from the OS and interrupts.

Note: The only impact of the normal and exception modes is on the selection of the active stack pointer.

5.3.1.1 Stack Memory Problem

In a single stack system, each task stack must allocate memory for OS function calls and interrupts. This leads to extra memory being allocated on each task stack. This is shown in Figure 5-5.

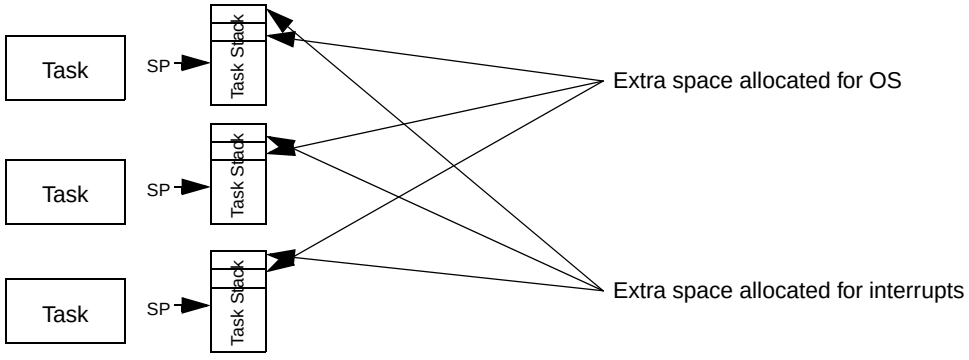


Figure 5-5. Single Stack Multitasking Memory Use

The memory space for OS calls and interrupts is replicated on each task stack. This situation is apparent if one considers that any task can call the OS functions. The OS functions can use the stack for subroutines, local variables, and so on. Since any task can make these calls, each task stack must be increased by the size of the OS memory use.

Memory space is required for interrupts because any task may be active when an interrupt occurs. The ISR pushes registers on the current stack and may also allocate local variables on the current stack. Since it is not known which task is being executed when an interrupt occurs, each task stack must be increased by the maximum ISR memory use.

In both situations, the memory is used only once, but it is allocated in more than one location. OS functions return without switching tasks. OS calls are not preemptable, although they are interruptible. Interrupts have the same behavior as OS functions, namely interrupts return without switching tasks.

5.3.1.2 SC140 Stack Utilization Solution

The solution to excessive memory use is to separate tasks from the OS and interrupts. This is done by using a normal/exception mode. The programming model has two stack pointers, NSP and ESP. The NSP is used by tasks when the core is in the normal mode of processing. The ESP is used by the OS and interrupts.

Since the OS and interrupts have their own stack pointer, memory for the OS and interrupts can be allocated separately. This allows the OS and interrupts code to be modified independently of the tasks.

Figure 5-6 shows the stack structure.

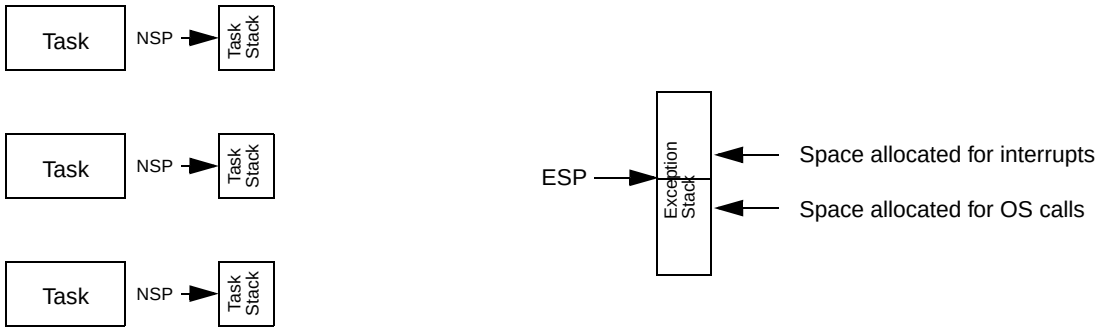


Figure 5-6. SC140 Memory Use with Multiple Stacks

The core changes to exception mode whenever it is processing an exception. When an exception occurs, the core switches to the ESP and uses the exception stack for saving registers and allocating local variables and subroutine calls.

OS calls are performed by executing a TRAP instruction, which generates a software interrupt. Since the processor is now in exception mode, all stack memory use is on the exception stack.

Once the core moves to exception mode, the OS usually needs to save the current context by stacking the registers in the normal stack. For this purpose, in addition to the regular push and pop instructions, specialized push and pop instructions (PUSHN/POPN) are provided that always access the normal stack, regardless of the mode.

5.3.1.3 Switching between Normal and Exception Modes

The core can be in either normal mode or exception mode, depending on the state of the EXP bit in the SR. If EXP is set, the core is in exception mode. EXP is a read/write bit and is set after reset. It is set automatically when the core starts processing an exception. The EXP bit can be cleared, causing an exit from exception mode, under the following conditions:

- The core executes an RTE instruction and restores the SR from the stack, including the previous value of the EXP bit.
- The user specifically clears the bit by writing to the SR.

Some instructions use the stack pointer implicitly or explicitly. The identity of the stack pointer used by these instructions is determined by the EXP bit:

- In exception mode (EXP = 1), all references to the stack use the ESP.
- In normal mode (EXP = 0), all references to the stack use the NSP.
- TFRA to or from OSP. This is a transfer to or from the other stack pointer. If in normal mode (EXP = 0), the user can access the ESP, and if in exception mode (EXP = 1), the user can access the NSP.

5.3.2 Software Stack

The core has specialized push and pop instructions that reference the active stack pointer (NSP or ESP). Table 5-7 describes these instructions.

Table 5-7. Stack Push/Pop Instructions

Instruction	Description
PUSH	Push a single 32-bit register onto the active stack and increment pointer by eight
PUSHN	Same as PUSH, but using the NSP, regardless of the mode
POP	Pre-decrement the stack by eight and restore one 32-bit register
POPN	Same as POP, but using the NSP, regardless of the mode
TFRA OSP,R	Move the other SP to a register
TFRA R,OSP	Move a register to the other SP

In addition, the stack can be accessed with move or bit mask instructions that use short and word displacement addressing using the stack pointer as a base pointer. Generally, the stack pointer points to the next unoccupied location. While using the pop/push instructions, all SC140 registers are viewed as two separate banks, an even register file bank and an odd register file bank, as shown in Table 5-8.

Table 5-8. Even and Odd Registers

Even Register File	Odd Register File
D0 D2 D4 D6	D1 D3 D5 D7
D0.e D2.e D4.e D6.e	D1.e D3.e D5.e D7.e
D0.e:D1.e D2.e:D3.e D4.e:D5.e D6.e:D7.e	R1 R3 R5 R7
R0 R2 R4 R6	B1 B3 B5 B7
B0 B2 B4 B6	N1 N3 M1 M3
N0 N2 M0 M2	LC0 LC1 LC2 LC3
SA0 SA1 SA2 SA3	VBA PCTL1 SR MCTL
PCTL0	

Up to two push instructions are supported in a single execution set. If two push instructions are included in a single execution set, the first push must use an even register operand and the second push must use an odd register operand. A push instruction always pushes one 32-bit register into the stack. Any execution set that includes one or two push instructions increments the stack pointer by eight. In the case of a single push, a single operand is written to the memory while the adjacent memory location remains unused.

Table 5-9 describes the stack memory map while performing a single or a dual push access.

Table 5-9. Stack Memory Map

Type	Memory Location X+4	Memory Location X
Single push - even register	Unused	Even operand
Single push - odd register	Odd operand	Unused
Dual push	Odd operand	Even operand

Similarly, up to two pop instructions are supported in a single execution set. If two pop instructions are included in a single execution set, the first pop must use an even register operand and the second pop must use an odd register operand.

An execution set which includes one or two pop instructions pre-decrements the stack pointer by eight, then restores one or two registers, as specified by the operands. Note that if the stack is popped with one register only, the other pushed register data may be lost.

Pushing and popping the data extension register (Dx.e + Lx tag bit) are unique operations. It is possible to push two extensions that are coupled together to form a single operand, or to push a single extension. The single extensions are divided between the even and odd tables. In both cases, the push operation occupies 32 bits. For more information, see Table 5-8 on page 5-22.

In general, for correct operation, the stack should be popped with exactly the same register pairing used in the execution set with which it is pushed. When a dual-extension push instruction is performed, the corresponding pop should similarly be made into dual extensions. If a single extension push is performed, the pop should similarly be made into a single extension from the same parity table (even or odd, respectively).

In addition to the push and pop instructions, the user can access the stack directly with move or bit-mask instructions. The available addressing modes are shown in Table 5-10. Note that the user cannot use addressing modes that update SP during the access, but only short or word displacement addressing modes that leave the SP unchanged.

Table 5-10. Stack Move Instructions

Addressing Mode	Description
(SP - xx)	Subtract offset by a shifted unsigned 5-bit, or 6-bit immediate value, SP unchanged.
(SP + xxxx)	Add a signed 15-bit immediate offset, SP unchanged

5.3.3 Shadow Stack Pointer Registers

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

When an NSP or ESP is written (by TFRA), then its shadow register automatically becomes invalid, in which case, the first pop instruction takes an additional cycle. When a push/pop instruction is executed, then the shadow register of the active NSP or ESP becomes valid.

5.3.4 Fast Call-Return from Subroutines

The SC140 supports a mechanism for speeding up the execution of the return from subroutine (RTS) instruction, using a return address register (RAS). The RAS is updated with the return address during the execution of a JSR or BSR instruction.

While normal execution of an RTS takes five to six execution cycles, if the routine performing the RTS is a leaf routine, meaning if no other RTS has been executed between the jump to this subroutine and the execution of the RTS, the core is able to execute the RTS instruction in three cycles.

The RTSTK instruction can be used to force the core to bypass the special logic that implements this fast RTS mechanism. RTSTK is typically used when the return address from subroutine is explicitly changed in the stack.

5.4 Processing States

The SC140 core is always in one of the six processing states: Normal, Exception, Debug mode, Reset, Wait or Stop. These states are described in the sections that follow. In some states, the operation of peripherals and other blocks is affected. The descriptions of the change in operation given here may be different for certain products that utilize a SC140 core. Consult the product-specific manuals for details of actions in each processing state.

5.4.1 Normal Processing State

In the normal processing state, the instructions are fetched and executed sequentially unless there is a change of flow. The core stays in the normal processing state, unless:

- An exception is encountered, as described in Section 5.4.2, “Exception Processing State.”
- A debug request is encountered, as described in Section 5.4.4, “Debug Mode State.”
- A WAIT or STOP instruction is executed, as described in Section 5.4.5, “Wait Processing State.”
- A hardware reset occurs, as described in Section 5.4.3, “Reset Processing State.”

5.4.2 Exception Processing State

The SC140 core can enter the exception processing state in any of the following ways:

- An external hardware exception (namely, an interrupt) request is issued to the core by either an off-chip device, or an on-chip peripheral.
- A software exception request (TRAP) is issued by the program itself.
- An internal system error, namely an illegal opcode, illegal execution set, or Data ALU overflow occurs.
- A debug interrupt request is issued by the EOnCE.

In all these cases, the SC140 core interrupts the normal program execution with an exception. When the core starts servicing the exception request, the following occurs:

1. The EXP bit in the status register is set, and the ESP becomes active instead of the NSP.
2. The PC is pushed to the software stack, along with the previous SR, at the position pointed to by the ESP.
3. The PC is loaded with a new value, as determined by the source of the exception request.

This is the default state of the core after exiting the reset state. Refer to Section 5.5, “Exception Processing,” for a detailed description of the exception processing state, and of the way an exception is serviced.

5.4.3 Reset Processing State

The reset processing state is entered when a core hardware reset occurs. Upon entering the reset state, the following registers are updated with their reset values:

- SR
- EMR
- VBA
- MCTL
- PCTL0
- PCTL1

Refer to Chapter 3, “Control Registers,” for details on reset of the various bits of the above registers.

The core remains in the reset state until the end of hardware reset. Upon leaving the reset state, the core enters the exception processing state, and program execution begins at the program memory address, which is derivative-dependent.

5.4.4 Debug Mode State

The Debug mode is a special core processing state in which the pipeline is stalled and waiting for user commands from the JTAG or EOnCE. The core can enter the Debug mode in the following cases:

- JTAG issues a debug request.
- The EE0 EOnCE pin is asserted during reset, or if programmed before, as a debug request input.
- An EOnCE Debug mode event occurs.
- A DEBUG instruction is executed.

The Debug mode is exited by setting the Exit bit in the EOnCE command register. Refer to Chapter 4, “Emulation and Debug (EOnCE),” for a detailed description of the user commands in the Debug mode.

5.4.5 Wait Processing State

The wait processing state is a low power consumption state entered by execution of the WAIT instruction. It takes ten core clock cycles from the issue of the WAIT instruction to enter the wait state when the global clock to the entire core is turned off. Peripherals can continue to operate, but all internal processing is halted until one of the following actions occurs:

- An enabled interrupt is issued.
- A non-maskable interrupt (NMI) request is issued.
- A low level is applied to the RESET pin (RESET asserted).
- The JTAG issues a debug request.
- The EE0 pin, programmed as a debug request input, is asserted.

If an exit from the wait state is caused by a high level on the EE0 pin or by a debug request, and the IME bit in the EOnCE EMCR register is not set, the core enters the Debug mode immediately. If the IME bit is set, the Debug exception is serviced instead of entering Debug mode.

Table 5-11 shows the wait process under various core conditions.

Table 5-11. Wait Processing

IPL	DI	Wait Process
Request IPL > core IPL as determined by the I2–I0 bits of the SR	Clear (interrupts enabled)	Exit the wait processing state. Jump to the ISR.
Request IPL > core IPL as determined by the I2–I0 bits of the SR	Set (interrupts disabled)	Exit the wait processing state. Execute the following execution sets. No jump to the ISR.
Request IPL <= core IPL as determined by the I2–I0 bits of the SR	Clear or set	Remain in the wait processing state.
An NMI is asserted	Clear or set	Exit the wait processing state. Jump to the ISR.

5.4.6 Stop Processing State

The stop processing state is the lowest power consumption mode and is entered by execution of the STOP instruction. It takes ten core clock cycles from the issue of the STOP instruction to enter the stop state when the global clocks to the entire core and peripherals are turned off.

All activity in the core is halted until one of the following actions occurs:

- A low level is applied to an external dedicated pin.
- A low level is applied to the RESET pin (RESET asserted).
- The JTAG issues a debug request.
- The EE0 pin, programmed as a debug request input, is asserted.

Any of these actions turns on the oscillator and after a clock stabilization delay, clocks to the core are re-enabled.

If the stop processing state is exited by assertion of the RESET pin, the core enters the reset processing state.

If the stop processing state is exited during the assertion of an external interrupt request, the core services the highest priority pending interrupt. If no interrupt is pending, the core resumes execution of the instruction following the STOP instruction that caused the entry into the stop state.

If the stop processing state is exited by a high level on the EE0 pin or by a debug request from the JTAG, and the IME bit in the EOnCE EMCR register is not set, the core enters Debug mode immediately. If the IME bit is set, the Debug exception is serviced instead of entering Debug mode.