# 4 PROGRAM SEQUENCER

In the ADSP-21535 DSP, the Program Sequencer controls program flow, constantly providing the address of the next instruction to be executed by other parts of the ADSP-21535 processor. Program flow in the chip is mostly linear, with the processor executing program instructions sequentially.

The linear flow varies occasionally when the program uses nonsequential program structures, such as those illustrated in Figure 4-1. Nonsequential structures direct the ADSP-21535 DSP to execute an instruction that is not at the next sequential address. These structures include:

- **Loops.** One sequence of instructions executes several times with zero overhead.
- **Subroutines.** The processor temporarily interrupts sequential flow to execute instructions from another part of memory.
- **Jumps.** Program flow transfers permanently to another part of memory.
- **Interrupts and Exceptions.** A runtime event or instruction triggers the execution of a subroutine.

- **Idle**. An instruction causes the processor to stop operating and hold its current state until an interrupt occurs. Then, the processor services the interrupt and continues normal execution.

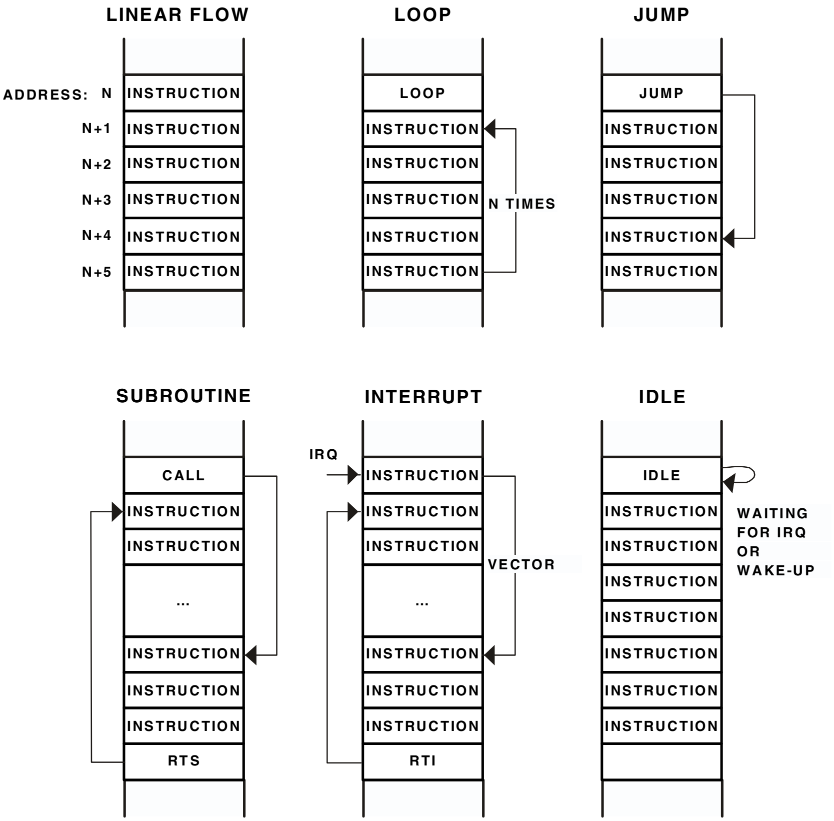


Figure 4-1. Program Flow Variations

The Sequencer manages execution of these program structures by selecting the address of the next instruction to execute.

The fetched address enters the instruction pipeline, ending with the program counter (PC). The pipeline contains the 32-bit addresses of the instructions currently being fetched, decoded, and executed. The PC couples with the RETn registers, which store return addresses. All addresses generated by the Sequencer are 32-bit memory instruction addresses.

To manage events, the Sequencer's event controller handles interrupt and event processing, determines whether an interrupt is masked, and generates the appropriate event vector address.

In addition to providing data addresses, the Data Address Generators (DAGs) can provide instruction addresses for the Sequencer's indirect branches.

The Sequencer evaluates conditional instructions and loop termination conditions. The loop registers support nested loops. The memory-mapped registers (MMRs) store information used to implement interrupt service routines.

# Sequencer Related Registers

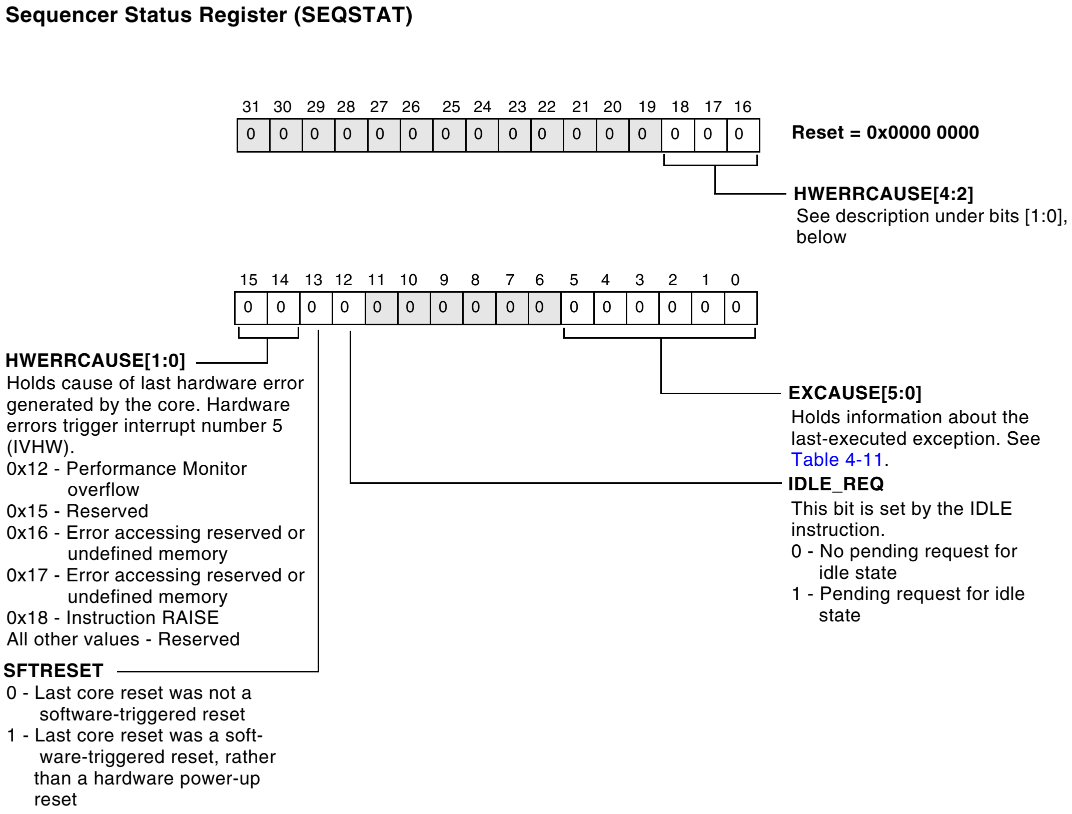
Table 4-1 lists the registers within the ADSP-21535 processor that are related to the Sequencer. Except for the PC register, all Sequencer related registers are directly readable and writable. Manually pushing or popping registers to or from the stack is done using the explicit instructions `[--SP] = Rn` (for push) or `Rn = [SP++]` (for pop).

Table 4-1. Sequencer Related Registers

| Register Name | Description |
|---|---|
| SEQSTAT | Sequencer Status register |
| | Return Address registers: See "Events and Sequencing" on page 4-18. |
| RETX<br>RETN<br>RETI<br>RETE<br>RETS | Exception Return<br>NMI Return<br>Interrupt Return<br>Emulation Return<br>Subroutine Return |
| | Zero-Overhead Loop registers: |
| LC0, LC1<br>LT0, LT1<br>LB0, LB1 | Loop Counters<br>Loop Tops<br>Loop Bottoms |
| FP, SP | Frame Pointer and Stack Pointer: See "Data Address Generators" on page 5-1. |
| SYSCFG | System Configuration Register |
| CYCLES, CYCLES2 | Cycle Counters: See "Blackfin DSP's Debug" on page 20-1. |

## Sequencer Status Register (SEQSTAT)

The Sequencer Status register (`SEQSTAT`), shown in Figure 4-2, contains information about the current state of the Sequencer as well as diagnostic information from the last event. `SEQSTAT` is a read-only register and is accessible only in Supervisor mode.



**HWERRCAUSE[4:2]**
See description under bits [1:0], below

**HWERRCAUSE[1:0]**
Holds cause of last hardware error generated by the core. Hardware errors trigger interrupt number 5 (IVHW).
0x12 - Performance Monitor overflow
0x15 - Reserved
0x16 - Error accessing reserved or undefined memory
0x17 - Error accessing reserved or undefined memory
0x18 - Instruction RAISE
All other values - Reserved

**SFTRESET**
0 - Last core reset was not a software-triggered reset
1 - Last core reset was a software-triggered reset, rather than a hardware power-up reset

**EXCAUSE[5:0]**
Holds information about the last-executed exception. See Table 4-11.

**IDLE_REQ**
This bit is set by the IDLE instruction.
0 - No pending request for idle state
1 - Pending request for idle state

Figure 4-2. Sequencer Status Register

## Zero-overhead Loop Registers (LC, LT, LB)

Two sets of zero-overhead loop registers implement loops, using hardware counters instead of software instructions to evaluate loop conditions. After evaluation, processing branches to a new target address. Both sets of registers include the Loop Counter (LC), Loop Top (LT), and Loop Bottom (LB) registers.

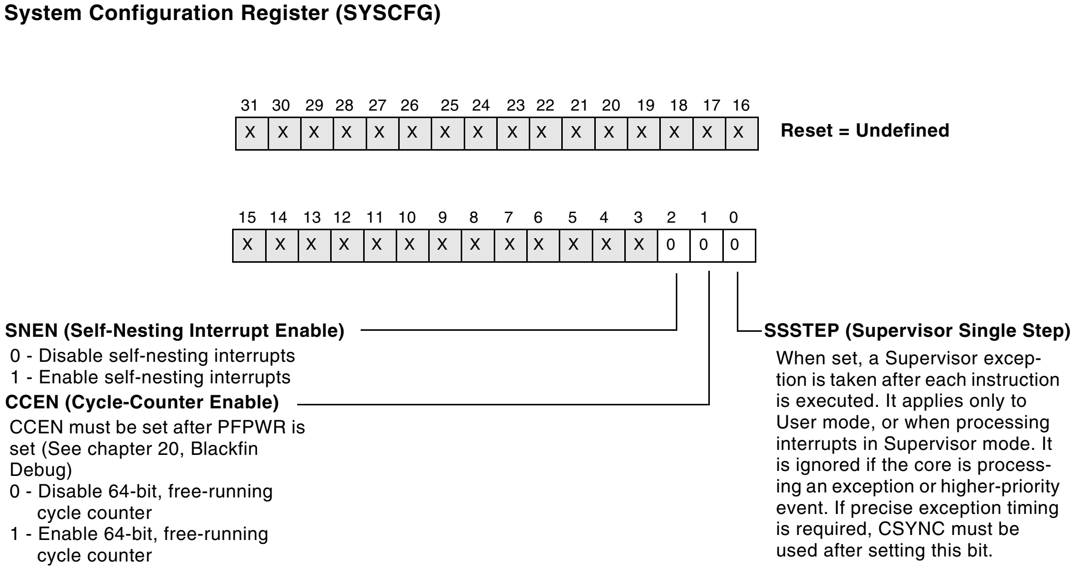
The 32-bit loop register sets are described in Table 4-2:

Table 4-2. Loop Registers

| Registers | Description | Function |
|---|---|---|
| LC[1:0] | Loop Counters | Maintain a count of the remaining iterations of the loop |
| LT[1:0] | Loop Tops | Hold the address of the first statement within a loop |
| LB[1:0] | Loop Bottoms | Hold the address of the last statement of the loop |

## System Configuration Register (SYSCFG)

The System Configuration Register (SYSCFG), shown in Figure 4-3, controls the configuration of the processor. This register is accessible only from the Supervisor mode.



**SNEN (Self-Nesting Interrupt Enable)**
0 - Disable self-nesting interrupts
1 - Enable self-nesting interrupts

**CCEN (Cycle-Counter Enable)**
CCEN must be set after PFPWR is set (See chapter 20, Blackfin Debug)
0 - Disable 64-bit, free-running cycle counter
1 - Enable 64-bit, free-running cycle counter

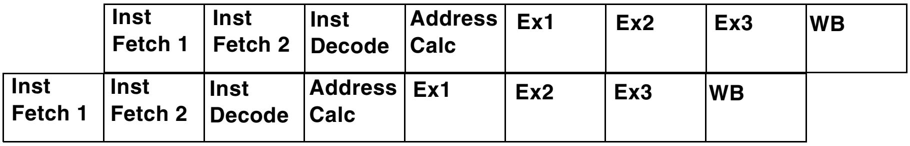
**SSSTEP (Supervisor Single Step)**
When set, a Supervisor exception is taken after each instruction is executed. It applies only to User mode, or when processing interrupts in Supervisor mode. It is ignored if the core is processing an exception or higher-priority event. If precise exception timing is required, CSYNC must be used after setting this bit.

Figure 4-3. System Configuration Register

# Instruction Pipeline

The Program Sequencer determines the next instruction address by examining both the current instruction being executed and the current state of the processor. If no conditions require otherwise, the DSP executes instructions from memory in sequential order by incrementing the look ahead address.

The ADSP-21535 processor has an eight-stage instruction pipeline, shown in Figure 4-4.



| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | Inst Fetch 1 | Inst Fetch 2 | Inst Decode | Address Calc | Ex1 | Ex2 | Ex3 | WB |
| Inst Fetch 1 | Inst Fetch 2 | Inst Decode | Address Calc | Ex1 | Ex2 | Ex3 | WB | |

Figure 4-4. ADSP-21535 DSP Pipeline

Table 4-3. Stages of Instruction Pipeline

| Pipeline Stage | Description |
|---|---|
| Instruction Fetch 1 (IF1) | Start instruction memory access. |
| Instruction Fetch 2 (IF2) | Finish L1 instruction memory access and align instruction. |
| Instruction Decode (DEC) | Start instruction decode and access Pointer register file. |
| Address Calculation (AC) | Calculate data addresses and branch target address. |
| Execute 1 (EX1) | Read data and start access of data memory. |
| Execute 2 (EX2) | Finish accesses of data memory and start execution of dual cycle instructions. |
| Execute 3 (EX3) | Execute single cycle instructions. |
| Write Back (WB) | Write states to Data and Pointer register files and process events. |

The Sequencer decodes and distributes operations to the Instruction Memory Unit and Instruction Alignment Unit. It also controls stalling and invalidating the instructions in the pipeline. The Sequencer ensures that the pipeline is fully interlocked and that the programmer does not need to manage the pipeline.

The instruction fetch and branch logic generates 32-bit fetch addresses for the Instruction Memory Unit. The Instruction Alignment Unit returns instructions and their width information at the end of the IF2 stage.

For each instruction type (16-, 32-, or 64-bit), the Alignment Unit ensures that the alignment buffers have enough valid data to be able to provide an instruction every cycle. Since the instructions can be 16, 32, or 64 bits wide, the Alignment Unit may not need to fetch data from the cache every cycle. For example, for a series of 16-bit instructions, the Alignment Unit gets data from the Instruction Memory Unit once in 4 cycles. The alignment logic requests the next instruction address based on the status of the alignment buffers. The Sequencer responds by generating the next fetch address in the next cycle, provided there is no change of flow.

The Sequencer holds the fetch address until it receives a request from the alignment logic or until a change of flow occurs. It always increments the previous fetch address by 8 (the next 8 bytes). If a change of flow occurs, such as a branch or an interrupt, the Sequencer communicates it to the Instruction Memory Unit, which invalidates the data in the Alignment Unit. In addition to the change-of-flow indication, the Sequencer can also kill instructions in IF1 and IF2 stages or stall the Instruction Memory Unit.

The Execution Unit contains two 16-bit multipliers, two 40-bit ALUs, two 40-bit accumulators, one 40-bit shifter, a video unit (which adds 8-bit ALU support), and an 8-entry 32-bit Data Register File.

Register File reads occur in the EX1 pipeline stage (for operands) and the EX3 pipeline stage (for stores). Writes occur in the WB stage. The multipliers and the video unit are active in the EX2 stage, and the ALUs and shifter are active in the EX3 stage. The accumulators are written at the end of the EX3 stage.

Any nonsequential program flow can potentially decrease the DSP's instruction throughput. Nonsequential program operations include:

- Program memory data accesses that conflict with instruction fetches
- Jumps
- Subroutine calls and returns
- Interrupts and returns
- Loops

# Branches and Sequencing

One type of nonsequential program flow that the Sequencer supports is branching. A branch occurs when a JUMP or CALL instruction begins execution at a new location other than the next sequential address. For descriptions of how to use the JUMP and CALL instructions, see the *Blackfin DSP Instruction Set Reference*. Briefly, these instructions operate as follows:

- A JUMP or a CALL instruction transfers program flow to another memory location. The difference between a JUMP and a CALL is that a CALL automatically loads the return address into the RETS register. The return address is the next sequential address after the CALL instruction. This push makes the address available for the CALL instruction's matching return instruction, allowing easy return from the subroutine.

- A return instruction causes the Sequencer to fetch the instruction at the return address, which is stored in the RETS register (for subroutine returns). The types of return instructions are return from subroutine (RTS), return from interrupt (RTI), return from exception (RTX), return from emulation (RTE), and return from non-maskable interrupt (RTN). Each return type has its own register for holding the return address.

- JUMP instructions can be conditional, depending on the status of the CC bit of the ASTAT register. They are immediate and may not be delayed. The Program Sequencer can evaluate the CC status bit to decide whether to execute a branch. If no condition is specified, the branch is always taken.
- Conditional JUMP instructions use static branch prediction to reduce the branch latency because of the effects of the pipeline.

Branches can be direct or indirect. The difference is that the Sequencer generates the address for a direct branch, for example JUMP 0x30, and the Data Address Generator produces the address for an indirect branch, for example JUMP (P3).

Direct branches are JUMP or CALL instructions that use a PC-relative address.

Indirect branches are JUMP or CALL instructions that use a dynamic address—an address that changes at runtime—that comes from a Data Address Generator. For more information, see “Data Address Generators” on page 5-1.

## Direct Short and Long Jumps

The Sequencer supports both short and long jumps. The target of the branch is a PC-relative address from the location of the instruction, plus an offset. The PC-relative offset for the short jump is a 13-bit immediate value that must be a multiple of two (bit zero must be a zero). The 13-bit value gives an effective dynamic range of –4096 to +4094 bytes.

The PC-relative offset for the long jump is a 25-bit immediate value that must also be a multiple of two (bit zero must be a zero). The 25-bit value gives an effective dynamic range of –16,777,216 to +16,777,214 bytes.

If, at the time of writing the program, the destination is known to be less than a 13-bit offset from the current PC value, then the `JUMP.S 0xnnnn` instruction may be used. If the destination requires more than a 13-bit offset, then the `JUMP.L 0xnnnn` instruction must be used. If the destination offset is unknown and development tools must evaluate the offset, then use the instruction `JUMP 0xnnnn`. Upon disassembly, the instruction is replaced by the appropriate `JUMP.S` or `JUMP.L` instruction.

## Direct Call

The `CALL` instruction is a branch instruction that copies the location of the instruction after the call into the `RETS` register. The `CALL` instruction has a 25-bit PC-relative offset that must be a multiple of two (bit zero must be a zero). The 25-bit value gives an effective dynamic range of –16,777,216 to +16,777,214 bytes.

## Indirect Branch and Call

The indirect `JUMP` and `CALL` instructions use a dynamic address—an address that changes at runtime—that comes from the Data Address Generator. The effective address is stored in a P-register. For the `CALL` instruction, the `RETS` register is loaded with the address of the instruction following the `CALL` instruction.

For example:

```
JUMP (P3) ;
CALL (P0) ;
```

## PC-Relative Indirect Branch and Call

The PC-relative indirect JUMP and CALL instructions use a P-register as an offset to the branch target. For the CALL instruction, the RETS register is loaded with the address of the instruction following the CALL instruction.

For example:

```
JUMP (PC + P3) ;
CALL (PC + P0) ;
```

## Condition Code Flag

The ADSP-21535 supports a condition code (CC) flag bit, which is used to resolve the direction of a branch. This flag may be accessed five ways:

- A conditional branch is resolved by the value in CC.
- A Data register value may be copied into CC, and the value in CC may be copied to a Data register.
- A status flag may be copied into CC, and the value in CC may be copied to a status flag.
- CC may be set to the result of a Pointer register comparison.
- CC may be set to the result of a Data register comparison.

These five ways of accessing the CC bit are used to control program flow. The branch is explicitly separated from the instruction that sets the arithmetic flags. A single bit resides in the instruction encoding that specifies the interpretation for the value of CC. The interpretation is branch on true or false.

The comparison operations have the form CC = expr where *expr* involves a pair of registers of the same type (for example, Data registers or Pointer registers, or a single register and a small immediate constant). The small immediate constant is a 3-bit (–4 through 3) signed number for signed comparisons and a 3-bit (0 through 7) unsigned number for unsigned comparisons.

The sense of CC is determined by equal (==), less than (<), and less than or equal to (<=). There are also bit test operations that test whether a bit in a 32-bit register is set.

## Conditional Branches

The Sequencer supports conditional branches. These are JUMP instructions whose execution is based on testing an IF condition that is based on the status of the CC bit. The target of the branch is a PC-relative address from the location of the instruction plus an offset. The PC-relative offset is an 11-bit immediate value that must be a multiple of two (bit zero must be a zero). This gives an effective dynamic range of –1024 to +1022 bytes.

For example, the following instruction tests the CC flag and, if it is positive, jumps to a location identified by the label dest_address.

```
IF CC JUMP dest_address ;
```

## Conditional Register Move

For condition handling where the value of a register is set based on the condition code, a conditional move instruction can be used instead of a branch statement. A register move can be performed, depending on whether the value of the CC flag is true or false (1 or 0). In some cases, using this instruction instead of a branch eliminates the cycles lost because of the branch.

Example code:

```
IF CC R0 = P0 ;
```

## Branch Prediction

The Sequencer supports static branch prediction to accelerate execution of conditional branches. These branches are executed based on the state of the CC bit.

The branch target address calculation takes place in the AC stage of the instruction pipeline. In the EX3 stage, the Sequencer compares the actual CC bit value to the predicted value. If the value was mis-predicted, the branch is corrected, and the correct address is available for the WB stage of the pipeline.

The branch latency for conditional branches is as follows:

- If prediction was “not to take branch,” and branch was actually not taken: 0 CCLK cycles.
- If prediction was “not to take branch,” and branch was actually taken: 6 CCLK cycles.
- If prediction was “to take branch,” and branch was actually taken: 3 CCLK cycles.
- If prediction was “to take branch,” and branch was actually not taken: 6 CCLK cycles.

For all unconditional branches, the branch target address computed in the AC stage of the pipeline is sent to the Instruction Fetch address bus at the beginning of the EX1 stage. All unconditional branches have a latency of 3 CCLK cycles.

Consider the two examples in Table 4-4.

Table 4-4. Branch Prediction

| Instruction | Description |
|---|---|
| `If CC JUMP back_dest (bp)` | This instruction tests the `CC` flag, and if it is set, jumps to a location, identified by the label, `back_dest`.<br>If `back_dest` is a prior address and the `CC` flag is set, then the branch is correctly predicted and the branch latency is reduced. |
| `If CC JUMP sub_dest (bp)` | This instruction tests the `CC` flag, and if it is set, jumps to a location, identified by the label, `sub_dest`.<br>If `sub_dest` is a subsequent address and the `CC` flag is set, then the branch is incorrectly predicted and the branch latency increases. |

# Loops and Sequencing

Another type of nonsequential program flow that the Sequencer supports is looping. A loop occurs when the values in a pair of loop top and loop bottom registers define an address range that includes the currently executing instruction. The corresponding loop counter must contain a nonzero value. One way to load these registers is by using the Loop Setup (`LSETUP`) instruction; however, it is also possible to load these registers directly. The ADSP-21535 DSP provides two sets of dedicated registers to support two nested loops.

The condition for terminating a loop is that the counter decreases to zero. This condition tests whether the loop has completed the number of iterations loaded from the Loop Count register (`LC[1]` or `LC[0]`).

The zero-overhead loop hardware provides efficient loops without the overhead of additional instructions to branch, test a condition, or decrement a counter. If the effective range of the loop needs to be larger than that shown in Table 4-5, the loop can be set up manually by loading the LCx, LTx, and LBx registers manually.

The code example in Listing 4-1 shows a loop that contains two instructions and iterates 32 times.

Listing 4-1. Loop

```
P5 = 0x20 ;
LSETUP ( lp_start, lp_end ) LC0 = P5 ;
lp_start:
R5 = R0 + R1 || R2 = [P2++] || R3 = [I1++] ;

lp_end:  R5 = R5 + R2 ;
```

Two sets of loop registers are used to manage two nested loops:

- LC[1:0] – the Loop Count registers
- LT[1:0] – the Loop Top address registers
- LB[1:0] – the Loop Bottom address registers

When executing an LSETUP instruction, the Program Sequencer loads the address of the loop's last instruction into LBx and the address of the loop's first instruction into LTx. The top and bottom addresses of the loop are computed as PC-relative addresses from the LSETUP instruction plus an offset. In each case, the offset value is added to the location of the LSETUP instruction.

`LC0` and `LC1` are unsigned 32-bit registers supporting $2^{32}-1$ iterations through the loop.

When `LCx = 0`, the loop is disabled, and a single pass of the code executes.

Table 4-5. Loop Registers

| First/Last Address of the Loop | PC-Relative Offset Used to Compute the Loop Start Address | Effective Range of the Loop Start Instruction |
|---|---|---|
| Top / First | 5-bit signed immediate; must be a multiple of 2. | 0 to 30 bytes away from `LSETUP` instruction. |
| Bottom / Last | 11-bit signed immediate; must be a multiple of 2. | 0 to 2046 bytes away from `LSETUP` instruction (the defined loop can be 2046 bytes long). |

The ADSP-21535 DSP supports a four-location instruction loop buffer that reduces instruction fetches while in loops. If the loop code contains four or fewer instructions, then no fetches to instruction memory are necessary for any number of loop iterations, because the instructions are stored locally. The loop buffer effectively shortens the instruction fetch time in loops with more than four instructions by allowing fetches to take place while instructions in the loop buffer are being executed.

# Events and Sequencing

The Event Controller of the processor manages five types of activities:

- Emulation
- Reset
- Non-maskable interrupts (NMI)

- Exceptions
- Interrupts

Note that the word *event* describes all five types. The Event Controller manages fifteen events in all: Emulation, Reset, NMI, Exception, and eleven interrupts.

An interrupt is an event that changes normal processor instruction flow and is asynchronous to program flow. In contrast, an exception is a software initiated event whose effects are synchronous to program flow.

The event system is nested and prioritized. Consequently, several service routines may be active at any time, and a low priority event is preempted by one of higher priority.

The ADSP-21535 DSP employs a two-level event control mechanism. The ADSP-21535 DSP System Interrupt Controller (SIC) works with the Core Event Controller (CEC) to prioritize and control all system interrupts. The SIC provides mapping between the many peripheral interrupt sources and the prioritized general-purpose interrupt inputs of the core. This mapping is programmable, and individual interrupt sources can be masked in the SIC.

The CEC supports nine general-purpose interrupts (IVG7 - IVG15) in addition to the dedicated interrupt and exception events that are described in Table 4-6. It is recommended that the lowest two priority interrupts (IVG14 and IVG15) be reserved for software interrupt handlers, leaving seven prioritized interrupt inputs (IVG7 - IVG13) to support the ADSP-21535 DSP system. Refer to Table 4-6.

Table 4-6. System and Core Event Mapping

| | Event Source | Core Event Name |
|---|---|---|
| Core Events | Emulation (highest priority) | EMU |
| | Reset | RST |
| | NMI | NMI |
| | Exception | EVX |
| | Reserved | – |
| | Hardware Error | IVHW |
| | Core Timer | IVTMR |
| System Interrupts | RTC<br>USB<br>PCI | IVG7 |
| | SPORT0 RX/TX<br>SPORT1 RX/TX | IVG8 |
| | SPI0<br>SPI1 | IVG9 |
| | UART0 RX/TX<br>UART1 RX/TX | IVG10 |
| | Timer0, Timer1, Timer2 | IVG11 |
| | Programmable Flags Interrupt A/B | IVG12 |
| | Memory DMA<br>Watchdog Timer | IVG13 |
| | Software Interrupt 1 | IVG14 |
| | Software Interrupt 2 (lowest priority) | IVG15 |

Note the System Interrupt to Core Event mappings shown are the default values at reset and can be changed by software.

## System Interrupt Processing

Referring to Figure 4-5 on page 4-23, note that when an interrupt (Interrupt A) is generated by an interrupt-enabled peripheral:

1. SIC_ISR logs the request and keeps track of system interrupts that are asserted but not yet serviced (that is, an interrupt service routine hasn't yet cleared the interrupt).
2. SIC_IWR checks to see if it should wake up the ADSP-21535 DSP core from an idled state based on this interrupt request.
3. SIC_IMASK masks off or enables interrupts from peripherals at the system level. If Interrupt A is not masked, the request proceeds to Step 4.
4. The SIC_IARx registers, which map the peripheral interrupts to a smaller set of general-purpose core interrupts (IVG7-IVG15), determine the core priority of Interrupt A.
5. ILAT adds Interrupt A to its log of interrupts latched by the core but not yet actively being serviced.
6. IMASK masks off or enables events of different core priorities. If the IVGx event corresponding to Interrupt A is not masked, the process proceeds to Step 7.
7. The Event Vector Table (EVT) is accessed to look up the appropriate vector for Interrupt A's interrupt service routine (ISR).

8. When the event vector for Interrupt A has entered the core pipeline, the appropriate IPEND bit is set, which clears the respective ILAT bit. Thus, IPEND tracks all pending interrupts, as well as those being presently serviced.

9. When the interrupt service routine for Interrupt A has been executed, the RTI instruction clears the appropriate IPEND bit. However, the relevant SIC_ISR bit is not cleared unless the interrupt service routine clears the mechanism that generated Interrupt A, or if the process of servicing the interrupt clears this bit.

It should be noted that emulation, reset, NMI, and exception events, as well as hardware error (IVHW) and core timer (IVTMR) interrupt requests, enter the interrupt processing chain at the ILAT level and are not affected by the system-level interrupt registers (SIC_IWR, SIC_ISR, SIC_IMASK, SIC_IARx).

If multiple interrupt sources share a single core interrupt, then the ISR must identify the peripheral that generated the interrupt. The ISR may then need to interrogate the peripheral to determine the appropriate action to take.

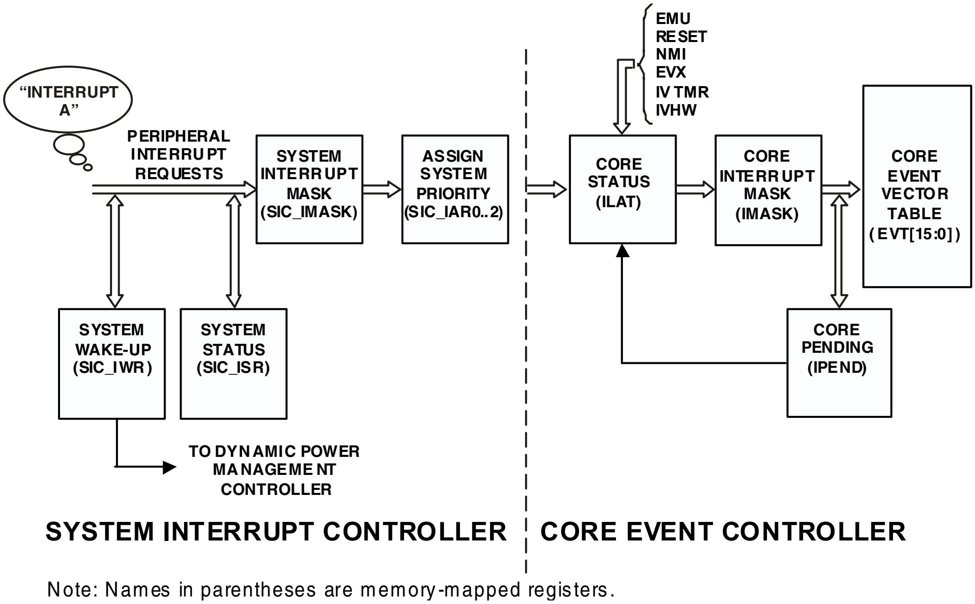

"INTERRUPT A"
PERIPHERAL INTERRUPT REQUESTS
SYSTEM INTERRUPT MASK (SIC_IMASK)
ASSIGN SYSTEM PRIORITY (SIC_IAR0..2)
SYSTEM WAKE-UP (SIC_IWR)
SYSTEM STATUS (SIC_ISR)
TO DYNAMIC POWER MANAGEMENT CONTROLLER
EMU
RESET
NMI
EVX
IVTMR
IVHW
CORE STATUS (ILAT)
CORE INTERRUPT MASK (IMASK)
CORE EVENT VECTOR TABLE (EVT[15:0])
CORE PENDING (IPEND)
SYSTEM INTERRUPT CONTROLLER
CORE EVENT CONTROLLER
Note: Names in parentheses are memory-mapped registers.

Figure 4-5. Interrupt Processing Block Diagram

## System Peripheral Interrupts

The ADSP-21535 DSP system has numerous peripherals, which therefore require many supporting interrupts. Table 4-7 lists:

- Peripheral interrupt source
- Peripheral interrupt ID used in the System Interrupt Assignment registers (SIC_IARx). See "System Interrupt Assignment Registers (SIC_IARx)" on page 4-31.

- General-purpose interrupt of the core to which the interrupt maps at reset
- The Core Interrupt ID used in the System Interrupt Assignment registers (SIC_IARx). See "System Interrupt Assignment Registers (SIC_IARx)" on page 4-31.

Table 4-7. Peripheral Interrupt Source Reset State

| Peripheral Interrupt Source | Peripheral Interrupt ID | General-purpose Interrupt (Assignment at Reset) | Core Interrupt ID |
|---|---|---|---|
| Real-Time Clock interrupts (alarm, second, minute, hour, countdown) | 0 | IVG7 | 0 |
| Reserved | 1 | IVG7 | 0 |
| USB interrupt | 2 | IVG7 | 0 |
| PCI interrupts | 3 | IVG7 | 0 |
| SPORT0 RX DMA interrupt | 4 | IVG8 | 1 |
| SPORT0 TX DMA interrupt | 5 | IVG8 | 1 |
| SPORT1 RX DMA interrupt | 6 | IVG8 | 1 |
| SPORT1 TX DMA interrupt | 7 | IVG8 | 1 |
| SPI0 DMA interrupt | 8 | IVG9 | 2 |
| SPI1 DMA interrupt | 9 | IVG9 | 2 |
| UART0 RX interrupt | 10 | IVG10 | 3 |
| UART0 TX interrupt | 11 | IVG10 | 3 |
| UART1 RX interrupt | 12 | IVG10 | 3 |
| UART1 TX interrupt | 13 | IVG10 | 3 |
| Timer0 interrupt | 14 | IVG11 | 4 |
| Timer1 interrupt | 15 | IVG11 | 4 |
| Timer2 interrupt | 16 | IVG11 | 4 |

Table 4-7. Peripheral Interrupt Source Reset State (Cont'd)

| Peripheral Interrupt Source | Peripheral Interrupt ID | General-purpose Interrupt (Assignment at Reset) | Core Interrupt ID |
|---|---|---|---|
| PF interrupt A | 17 | IVG12 | 5 |
| PF interrupt B | 18 | IVG12 | 5 |
| Memory DMA interrupt | 19 | IVG13 | 6 |
| Software Watchdog Timer interrupt | 20 | IVG13 | 6 |

The peripheral interrupt structure of the ADSP-21535 DSP is flexible. By default upon reset, multiple peripheral interrupts share a single, general-purpose interrupt in the core, as shown in the preceding table.

An interrupt service routine that supports multiple interrupt sources must interrogate the appropriate system MMRs to determine which peripheral generated the interrupt.

If the default assignments shown in Table 4-7 on page 4-24 are acceptable, then interrupt initialization involves only initialization of the core EVT vector address entries and IMASK register, and unmasking the specific peripheral interrupts in SIC_IMASK that the system requires.

## System Interrupt Wakeup-enable Register (SIC_IWR)

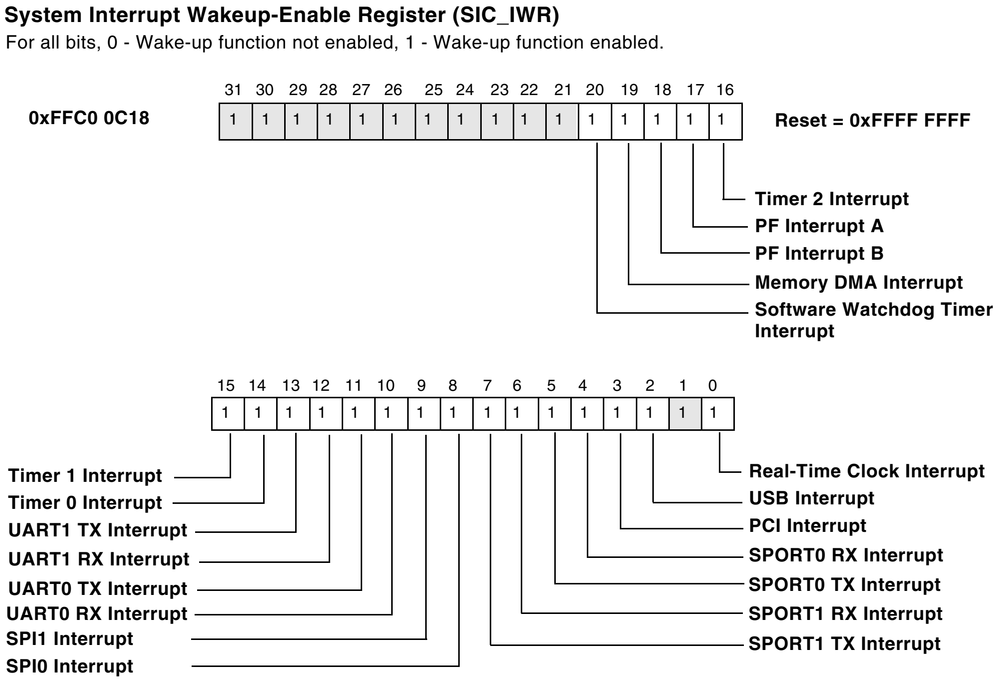
The SIC provides the mapping between the peripheral interrupt source and the Dynamic Power Management Controller (DPMC). Any of the ADSP-21535 DSP peripherals can be configured to wake up the core from its idled state to process the interrupt, simply by enabling the appropriate bit in the System Interrupt Wakeup-enable register (refer to Figure 4-6). If a peripheral interrupt source is enabled in SIC_IWR and the core is idled, the interrupt causes the DPMC to initiate the core wake-up sequence in order to process the interrupt. Note that this mode of operation may add latency to interrupt processing, depending on the power control state. For further discussion of power modes and the idled state of the core, see "Dynamic Power Management" on page 8-1.

By default, all interrupts generate a wake-up request to the core. However, for some applications it may be desirable to disable this function for some peripherals, such as a SPORTx Transmit Interrupt.

The SIC_IWR register has no effect unless the core is idled. The bits in this register correspond to those of the System Interrupt Mask (SIC_IMASK) and Interrupt Status (SIC_ISR) registers.

After reset, all valid bits of this register are set to 1, enabling the wake-up function for all interrupts that are not masked. Before enabling interrupts, configure this register in the reset initialization sequence. The SIC_IWR register can be read from or written to at any time. To prevent spurious or lost interrupt activity, this register should be written only when all peripheral interrupts are disabled.

Note the wake-up function is independent of the interrupt mask function. If an interrupt source is enabled in `SIC_ISR` but masked off in `SIC_IMASK`, the core wakes up if it is idled, but it does not generate an interrupt.


System Interrupt Wakeup-Enable Register (SIC_IWR)

For all bits, 0 - Wake-up function not enabled, 1 - Wake-up function enabled.

Figure 4-6. System Interrupt Wakeup-enable Register

## System Interrupt Status Register (SIC_ISR)

The SIC includes a read-only status register, the System Interrupt Status register, shown in Figure 4-7. Each valid bit in this register corresponds to one of the peripheral interrupt sources. The bit is set when the SIC detects the interrupt is asserted and cleared when the SIC detects that the peripheral interrupt input has been deasserted. Note that for some peripherals, such as programmable flag asynchronous input interrupts, many cycles of latency may pass from the time that an interrupt service routine initiates the clearing of the interrupt (usually by writing a system MMR) to the time that the SIC senses that the interrupt has been deasserted.

Depending on how interrupt sources map to the general-purpose interrupt inputs of the core, the interrupt service routine may have to interrogate multiple interrupt status bits to determine the source of the interrupt. One of the first instructions executed in an interrupt service routine should read SIC_ISR to determine whether more than one of the peripherals sharing the input has asserted its interrupt output. The service routine should fully process all pending, shared interrupts before executing the RTI, which enables further interrupt generation on that interrupt input.

> When an interrupt's service routine is finished, the RTI instruction clears the appropriate bit in the IPEND register. However, the relevant SIC_ISR bit is not cleared unless the service routine clears the mechanism that generated the interrupt.

Many systems need relatively few interrupt-enabled peripherals, allowing each peripheral to map to a unique core priority level. In these designs, SIC_ISR will seldom, if ever, need to be interrogated.

The SIC_ISR register is not affected by the state of the Interrupt Mask register and can be read at any time. Writes to SIC_ISR have no effect on its contents.

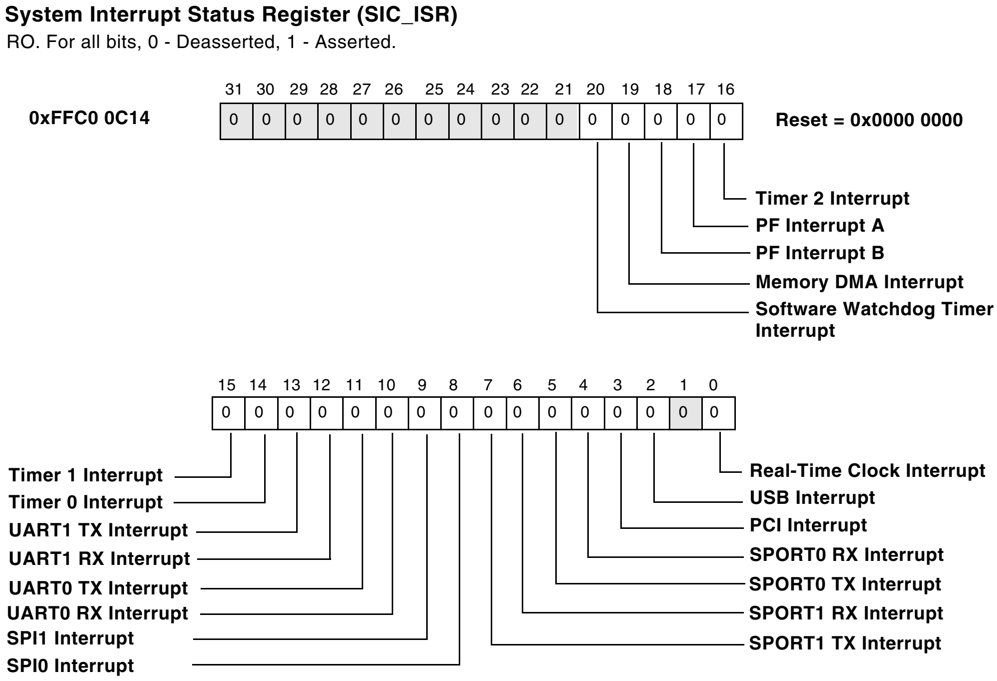


Figure 4-7. System Interrupt Status Register

## System Interrupt Mask Register (SIC_IMASK)

The System Interrupt Mask register, shown in Figure 4-8, allows masking of any peripheral interrupt source at the SIC, independently of whether it is enabled at the peripheral itself.

A reset forces the contents of `SIC_IMASK` to all 0s to mask off all peripheral interrupts. Writing a 1 to a bit location turns off the mask and enables the interrupt.

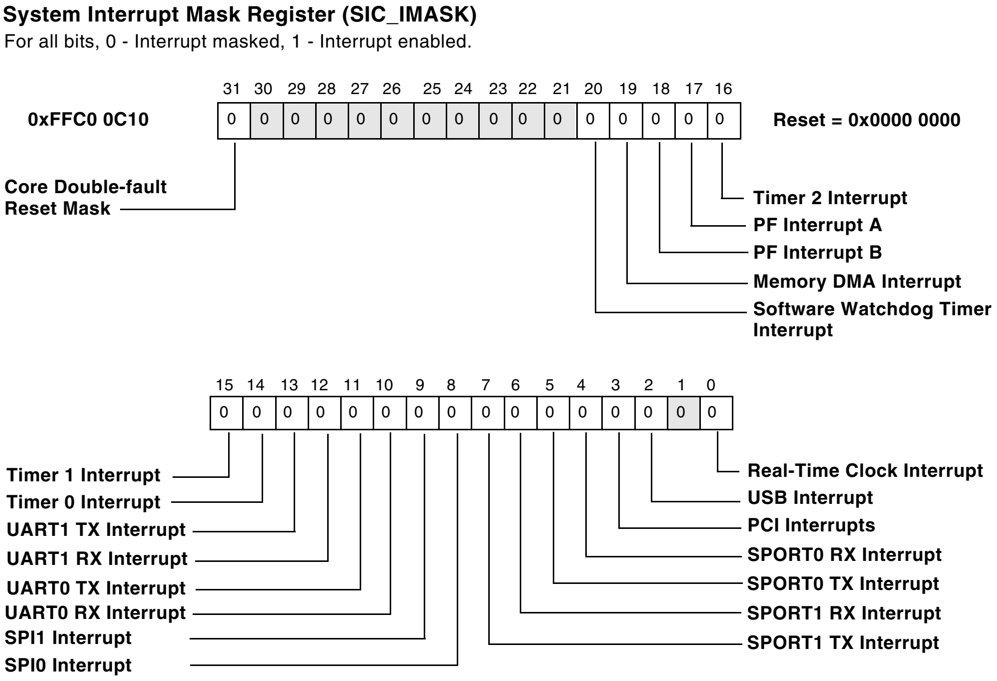


Figure 4-8. System Interrupt Mask Register

Although this register can be read from or written to at any time (in Supervisor mode), it should be configured in the reset initialization sequence before enabling interrupts.

SIC_IMASK[31] provides a mask for the core double-fault condition:

- If the condition is unmasked and the core detects a double-fault condition, a hardware reset is generated.
- If the condition is masked and the core detects a double-fault condition, core behavior may be unreliable.

## System Interrupt Assignment Registers (SIC_IARx)

The relative priority of peripheral interrupts can be set by mapping the peripheral interrupt to the appropriate general-purpose interrupt level in the core. The mapping is controlled by the System Interrupt Assignment register settings, as detailed in Figure 4-9, Figure 4-10, and Figure 4-11. If more than one interrupt source is mapped to the same interrupt, they are logically ORed, with no hardware prioritization. Software can prioritize the interrupt processing as required for a particular system application.

For general-purpose interrupts with multiple peripheral interrupts assigned to them, take special care to ensure that software correctly processes all pending interrupts sharing that input. Software is responsible for prioritizing the shared interrupts.

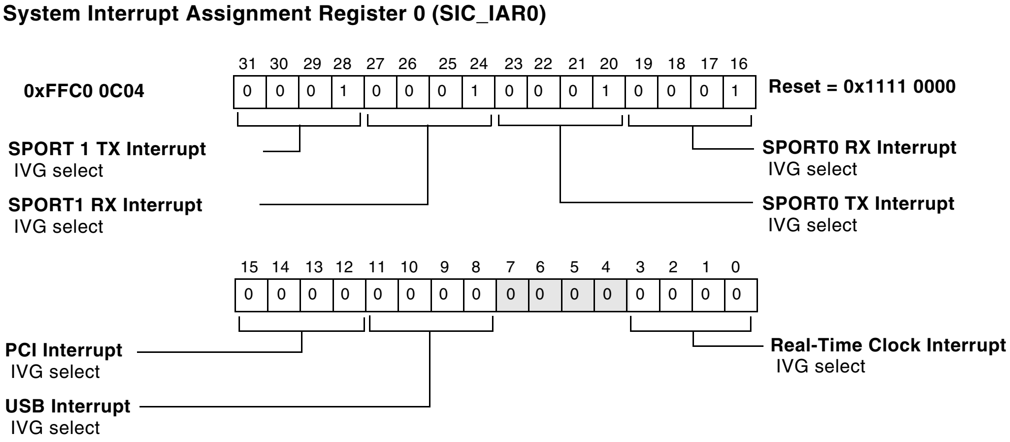


Figure 4-9. System Interrupt Assignment Register 0

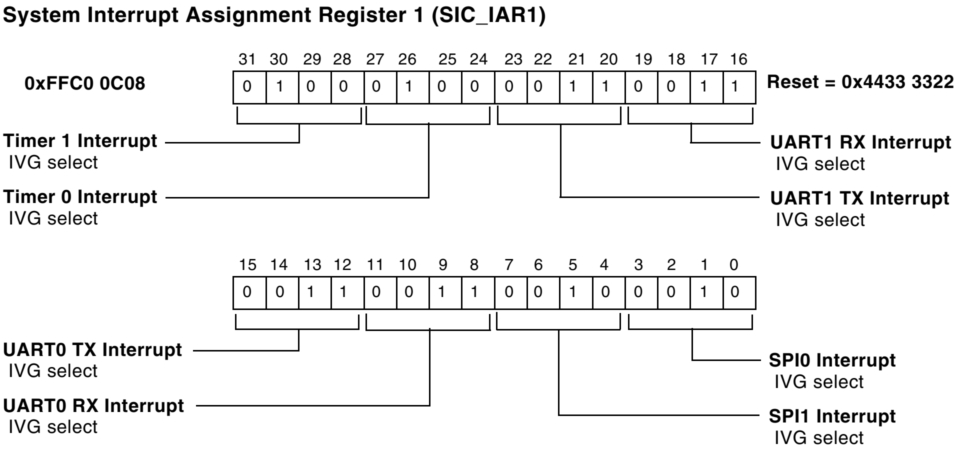

System Interrupt Assignment Register 1 (SIC_IAR1)
0xFFC0 0C08
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16
0 1 0 0 0 1 0 0 0 0 1 1 0 0 1 1
Reset = 0x4433 3322
Timer 1 Interrupt IVG select
Timer 0 Interrupt IVG select
UART1 RX Interrupt IVG select
UART1 TX Interrupt IVG select
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 1 1 0 0 1 1 0 0 1 0 0 0 1 0
UART0 TX Interrupt IVG select
UART0 RX Interrupt IVG select
SPI0 Interrupt IVG select
SPI1 Interrupt IVG select

Figure 4-10. System Interrupt Assignment Register 1

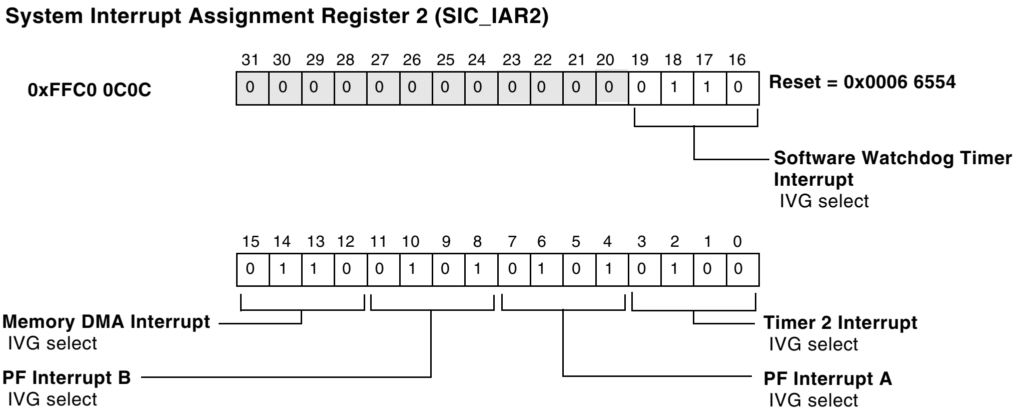

System Interrupt Assignment Register 2 (SIC_IAR2)
0xFFC0 0C0C
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16
0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 0
Reset = 0x0006 6554
Software Watchdog Timer Interrupt IVG select
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 1 1 0 0 1 0 1 0 1 0 1 0 1 0 0
Memory DMA Interrupt IVG select
PF Interrupt B IVG select
Timer 2 Interrupt IVG select
PF Interrupt A IVG select

Figure 4-11. System Interrupt Assignment Register 2

These registers can be read or written at any time in Supervisor mode. It is advisable, however, to configure them in the Reset interrupt service routine before enabling interrupts. To prevent spurious or lost interrupt activity, these registers should be written to only when all peripheral interrupts are disabled.

Table 4-8 defines the value to write in SIC_IARx to configure a peripheral for a particular IVG priority.

Table 4-8. IVG-Select Definitions

| General-purpose Interrupt | Value in SIC_IAR |
|---|---|
| IVG7 | 0 |
| IVG8 | 1 |
| IVG9 | 2 |
| IVG10 | 3 |
| IVG11 | 4 |
| IVG12 | 5 |
| IVG13 | 6 |
| IVG14 | 7 |
| IVG15 | 8 |

# Event Controller Registers

The Event Controller uses three MMRs to coordinate pending event requests. Each register is 16 bits wide, and each bit, N, corresponds to the EVT[N] event in the Event Vector Table. The registers are:

- IMASK - interrupt mask
- ILAT - interrupt latch
- IPEND - interrupts pending

The Event Controller updates ILAT and IPEND. The IPEND register is read-only in Supervisor mode. The ILAT and IMASK registers may be both read and written in Supervisor mode, with the exception of ILAT[0], which is read-only. None of the three registers can be accessed in User mode.

## Core Interrupt Mask Register (IMASK)

Each bit in IMASK indicates when the corresponding interrupt is enabled (see Figure 4-12). When an interrupt bit is set in the ILAT register, the corresponding interrupt is accepted only if the corresponding bit is also set in the IMASK register. The IMASK register may be read and written in Supervisor mode.

Only the emulator may mask emulation events. Please refer to "Blackfin DSP's Debug" on page 20-1.

**Core Interrupt Mask Register (IMASK)**

For all bits, 0 - Interrupt masked, 1 - Interrupt enabled.

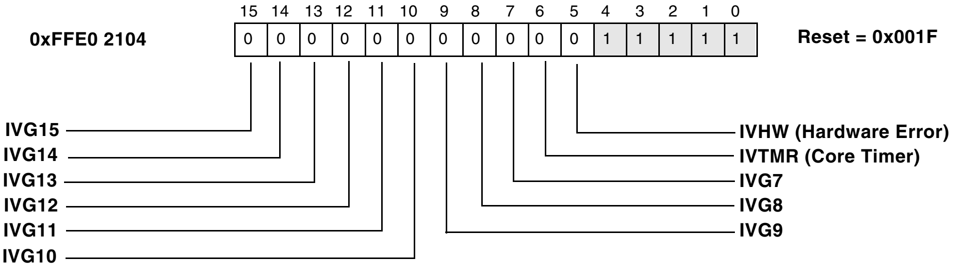


Figure 4-12. Core Interrupt Mask Register

## Core Interrupt Latch Register (ILAT)

Each bit in ILAT indicates when the corresponding event is latched (see Figure 4-13). The bit is reset before the first instruction in the corresponding ISR is executed. Writes to any ILAT bit can only occur when the corresponding IMASK bit is set. Otherwise, the write is ignored. This write functionality to ILAT is provided for cases where latched interrupt requests need to be cleared (cancelled) instead of servicing them.

When an event is serviced, its corresponding bit in ILAT is cleared. The RAISE N instruction causes specific bits in ILAT to be set and can trigger only events IVG15-IVG7, IVTMR, IVHW, NMI and RST.

Only the JTAG TRST pin can clear ILAT[0].

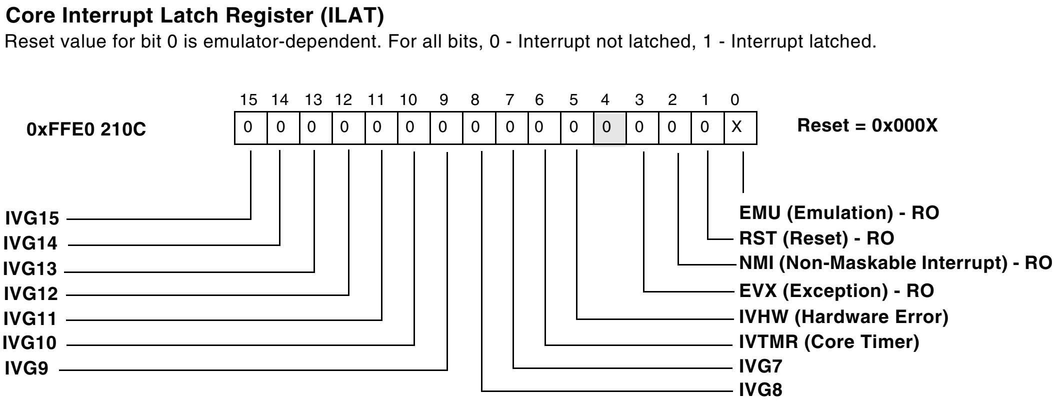
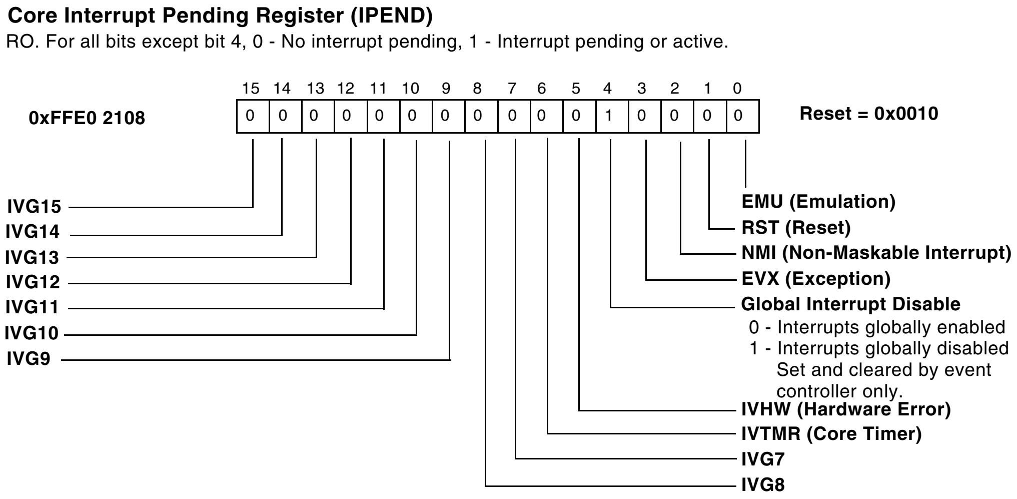

Core Interrupt Latch Register (ILAT)
Reset value for bit 0 is emulator-dependent. For all bits, 0 - Interrupt not latched, 1 - Interrupt latched.
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0xFFE0 210C
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 X
Reset = 0x000X
IVG15
IVG14
IVG13
IVG12
IVG11
IVG10
IVG9
EMU (Emulation) - RO
RST (Reset) - RO
NMI (Non-Maskable Interrupt) - RO
EVX (Exception) - RO
IVHW (Hardware Error)
IVTMR (Core Timer)
IVG7
IVG8

Figure 4-13. Core Interrupt Latch Register

## Core Interrupts Pending Register (IPEND)

The IPEND register keeps track of all currently nested interrupts (see Figure 4-14). Each bit in IPEND indicates that the corresponding interrupt is currently active or nested at some level. It may be read in Supervisor mode, but not written. The IPEND[4] bit is used by the Event Controller to temporarily disable interrupts on entry and exit to an interrupt service routine.

When an event is processed, the corresponding bit in `IPEND` is set. The least significant bit in `IPEND` that is currently set indicates the interrupt that is currently being serviced. At any given time, `IPEND` holds the current status of all nested events.


Core Interrupt Pending Register (IPEND)

RO. For all bits except bit 4, 0 - No interrupt pending, 1 - Interrupt pending or active.

Figure 4-14. Core Interrupt Pending Register

## Global Enabling/Disabling of Interrupts

General-purpose interrupts can be globally disabled with the `CLI Dreg` instruction and re-enabled with the `STI Dreg` instruction, both of which are only available in Supervisor mode. Reset, NMI, emulation, and exception events cannot be globally disabled. Globally disabling interrupts clears `IMASK[15:5]` after saving `IMASK`'s current state. See "Enable Interrupts" and "Disable Interrupts" in the "External Event Management" chapter of the *Blackfin DSP Instruction Set Reference*.

When program code is too time critical to be delayed by an interrupt, disable general-purpose interrupts, but be sure to re-enable them at the conclusion of the code sequence.

# Event Vector Table

The Event Vector Table (EVT) is a hardware table with sixteen entries that are each 32 bits wide. The EVT contains an entry for each possible core event. Entries are accessed as MMRs, and each entry can be programmed at reset with the corresponding vector address for the interrupt service routine. When an event occurs, instruction fetch starts at the address location in the EVT entry for that event.

The ADSP-21535 DSP architecture allows unique addresses to be programmed into each of the interrupt vectors; that is, interrupt vectors are not determined by a fixed offset from an interrupt vector table base address. This approach minimizes latency by not requiring a long jump from the vector table to the actual ISR code.

Table 4-9 lists events by priority. Each event has a corresponding bit in the event state registers ILAT, IMASK, and IPEND.

Table 4-9. Core Event Vector Table

| Event Number | Event Class | Name | MMR Location | Notes |
|---|---|---|---|---|
| EVT0 | Emulation | EMU | 0xFFE0 2000 | Highest priority. Vector address is provided by JTAG. |
| EVT1 | Reset | RST | 0xFFE0 2004 | Read-only. |
| EVT2 | NMI | NMI | 0xFFE0 2008 | |
| EVT3 | Exception | EVX | 0xFFE0 200C | |
| EVT4 | Reserved | Reserved | 0xFFE0 2010 | Reserved vector. |
| EVT5 | Hardware Error | IVHW | 0xFFE0 2014 | |

Table 4-9. Core Event Vector Table  (Cont'd)

| Event Number | Event Class | Name | MMR Location | Notes |
|---|---|---|---|---|
| EVT6 | Core Timer | IVTMR | 0xFFE0 2018 | |
| EVT7 | Interrupt 7 | IVG7 | 0xFFE0 201C | |
| EVT8 | Interrupt 8 | IVG8 | 0xFFE0 2020 | |
| EVT9 | Interrupt 9 | IVG9 | 0xFFE0 2024 | |
| EVT10 | Interrupt 10 | IVG10 | 0xFFE0 2028 | |
| EVT11 | Interrupt 11 | IVG11 | 0xFFE0 202C | |
| EVT12 | Interrupt 12 | IVG12 | 0xFFE0 2030 | |
| EVT13 | Interrupt 13 | IVG13 | 0xFFE0 2034 | |
| EVT14 | Interrupt 14 | IVG14 | 0xFFE0 2038 | |
| EVT15 | Interrupt 15 | IVG15 | 0xFFE0 203C | Lowest priority. |

## Emulation

An emulation event causes the processor to enter Emulation mode, in which instructions are read from the JTAG interface. It is the highest priority interrupt to the core.

For detailed information about emulation, see "Blackfin DSP's Debug" on page 20-1.

## Reset

The reset interrupt (RST) can be initiated via the RESET pin or through expiration of the Watchdog timer. The EVT[1] register holds the address at which the processor begins execution after reset. This location differs from that of other interrupts in that its content is read-only. Writes to this address have no effect.

The core has an output that indicates that a double-fault has occurred. This is a non-recoverable state. The SIC (via SIC_IMASK) can be programmed to send a reset request if a double-fault condition is detected. Subsequently, the reset request forces a system reset for core and peripherals.

The reset vector is determined by the ADSP-21535 DSP system. It points to the start of the on-chip boot ROM, or to the start of external asynchronous memory, depending on the state of the BMODE[2:0] pins. Refer to Table 4-10.

Table 4-10. Reset Vector Addresses

| Boot Source | BMODE[2:0] | Execution Start Address |
|---|---|---|
| Bypass boot ROM; execute from 16-bit-wide external memory (Async Bank 0) | 000 | 0x2000 0000 |
| Use boot ROM to boot from 8-bit flash | 001 | 0xF000 0000 |
| Use boot ROM to configure and load boot code from SPI0 serial ROM (8-bit address range) | 010 | 0xF000 0000 |
| Use boot ROM to configure and load boot code from SPI0 serial ROM (16-bit address range) | 011 | 0xF000 0000 |
| Reserved | 100-111 | N/A |

If the BMODE[2:0] pins indicate either booting from flash or serial ROM, the reset vector points to the start of the internal boot ROM, where a small bootstrap kernel resides. The bootstrap code reads the System Reset Configuration register (SYSCR) to determine the value of the BMODE[2:0] pins, which determine the appropriate boot sequence. For information about the ADSP-21535 DSP boot ROM, see "Booting Methods" on page 3-18.

If the BMODE[2:0] pins indicate to bypass boot ROM, the reset vector points to the start of the external asynchronous memory region. In this mode, the internal boot ROM is not used. To support reads from this memory region, the external bus interface unit (EBIU) uses the default external memory configuration that results from hardware reset.

## NMI (Non-Maskable Interrupt)

The NMI entry is reserved for a non-maskable interrupt, which can be generated by the Watchdog timer or by the NMI input signal to the ADSP-21535 processor. An example of an event that requires immediate processor attention, and thus is appropriate as an NMI, is a power-down warning. During the execution of the NMI service routine, exceptions are suspended, as they have a lower priority than the NMI interrupt.

ⓘ If an exception occurs during handling of an NMI, servicing the exception is delayed until completion of the NMI.

## Exceptions

Exception events are synchronous to the instruction that generates the exception; that is, the exception is taken before the instruction is allowed to complete. Note, however, the architecture can generate exceptions for memory references that meet any of these criteria:

- Misalign
- Miss the ICPLB (for an instruction fetch), or DCPLB (for a load or store)
- Violate the protection specification for the CPLB entry
- Generate multiple hits on the respective CPLB
- Receive a bus error response on an instruction fetch

For more information about alignment and CPLBs, see "Memory" on page 6-1.

As shown in Table 4-11, exceptions can be services or error conditions. The value in the Type column indicates whether the exception for that row is a service or an error condition.

- For services (S), the return address is the address of the instruction that follows the exception, because a service is never re-executed.

  To allow the program to resume execution on return from a service routine, the processor state must be preserved before the excepting instruction.

- For error conditions (E), the return address is the address of the instruction that caused the exception, because some types of errors, such as a page fault, require that the offending instruction be re-executed.

  Determining whether the instruction is re-executed depends on the excepting instruction and error condition and is determined by the exception handler. When the instruction should be re-executed, the return address is used. When the instruction should not be re-executed, the exception handler must advance the return address by the instruction length.

- EXCAUSE[5:0] is placed in the Sequencer Status register (SEQSTAT), depending on the type of event.

Table 4-11. Events That Cause Exceptions

| **Exception** | **EXCAUSE[5:0]** | **Type:**<br>**(E) error**<br>**(S) service**<br>**See note 1.** | **Notes/Examples** |
|---|---|---|---|
| Force Exception instruction EXCPT with 4-bit field m | m-field | S | Instruction provides 4 bits of EXCAUSE. |
| Single step | 0x10 | S | When the processor is in single step mode, every instruction generates an exception. Primarily used for debugging. |
| Exception caused by an emulation trace buffer overflow | 0x11 | S | The processor takes this exception when the trace buffer overflows (only when enabled by the trace unit control register). |
| Undefined instruction | 0x21 | E | May be used to emulate instructions that are not defined for a particular processor implementation. |
| Illegal instruction combination | 0x22 | E | See section for multi-issue rules in the *Blackfin DSP Instruction Set Reference*. |

Table 4-11. Events That Cause Exceptions (Cont'd)

| Exception | EXCAUSE[5:0] | Type:<br>(E) error<br>(S) service<br>See note 1. | Notes/Examples |
|---|---|---|---|
| Data access CPLB protection violation | 0x23 | E | Attempted read or write to supervisor resource, or illegal data memory access. Supervisor resources are registers and instructions that are reserved for Supervisor use: Supervisor only registers, all MMRs, and Supervisor only instructions. (A simultaneous, dual access to two MMRs using the Data Address Generators generates this type of exception.) In addition, this entry is used to signal a protection violation caused by disallowed memory access, and it is defined by the Memory Management Unit (MMU) cacheability protection lookaside buffer (CPLB). |
| Data access misaligned address violation | 0x24 | E | Attempted misaligned data memory or data cache access. |
| Unrecoverable event | 0x25 | E | For example, an exception generated while processing a previous exception. |
| Data access CPLB miss | 0x26 | E | Used by the MMU to signal a CPLB miss on a data access. |
| Data access multiple CPLB hits | 0x27 | E | More than one CPLB entry matches data fetch address. |
| Exception caused by an emulation watchpoint match | 0x28 | E | There is a watchpoint match, and one of the EMUSW bits in the Watchpoint Instruction Address Control register (WPIACTL) is set. |
| Instruction fetch access exception | 0x29 | E | Error from instruction fetch, for example, instruction bus parity error. |

Table 4-11. Events That Cause Exceptions (Cont'd)

| Exception | EXCAUSE[5:0] | Type:<br>(E) error<br>(S) service<br>See note 1. | Notes/Examples |
|---|---|---|---|
| Instruction fetch misaligned address violation | 0x2A | E | Attempted misaligned instruction cache fetch. On a misaligned instruction fetch exception, the return address provided in RETX is the destination address which is misaligned, rather than the address of the offending instruction. For example, if an indirect branch to a misaligned address held in P0 is attempted, the return address in RETX is equal to P0, rather than to the address of the branch instruction. (Note that this exception can never be generated from PC-relative branches, only from indirect branches.) |
| Instruction fetch CPLB protection violation | 0x2B | E | Illegal instruction fetch access (memory protection violation). |
| Instruction fetch CPLB miss | 0x2C | E | CPLB miss on an instruction fetch. |
| Instruction fetch multiple CPLB hits | 0x2D | E | More than one CPLB entry matches instruction fetch address. |
| Illegal use of supervisor resource | 0x2E | E | Attempted to use a Supervisor register or instruction from User mode. Supervisor resources are registers and instructions that are reserved for Supervisor use: Supervisor only registers, all MMRs, and Supervisor only instructions. |

Note 1: For services (S), the return address is the address of the instruction that follows the exception. For errors (E), the return address is the address of the excepting instruction.

If an instruction causes multiple exceptions, only the exception with the highest priority is taken. Table 4-12 ranks exceptions by descending priority.

Table 4-12. Exceptions by Descending Priority

| Priority | Exception | EXCAUSE |
|---|---|---|
| 1 | Unrecoverable Event | 0x25 |
| 2 | I-Fetch Multiple CPLB Hits | 0x2D |
| 3 | I-Fetch Misaligned Access | 0x2A |
| 4 | I-Fetch Protection Violation | 0x2B |
| 5 | I-Fetch CPLB Miss | 0x2C |
| 6 | I-Fetch Access Exception | 0x29 |
| 7 | Watchpoint Match | 0x28 |
| 8 | Undefined Instruction | 0x21 |
| 9 | Illegal Combination | 0x22 |
| 10 | Illegal use protected resource | 0x2E |
| 11 | DAG0 Multiple CPLB Hits | 0x27 |
| 12 | DAG0 Misaligned Access | 0x24 |
| 13 | DAG0 Protection Violation | 0x23 |
| 14 | DAG0 CPLB Miss | 0x26 |
| 15 | DAG1 Multiple CPLB Hits | 0x27 |
| 16 | DAG1 Misaligned Access | 0x24 |
| 17 | DAG1 Protection Violation | 0x23 |
| 18 | DAG1 CPLB Miss | 0x26 |
| 19 | EXCPT instruction | m- field |
| 20 | Single Step | 0x10 |
| 21 | Trace Buffer | 0x11 |

## Exceptions While Executing an Exception Handler

While executing the exception handler, avoid issuing an instruction that generates another exception. Generating an exception before the exception handler finishes results in these actions:

- The generated exception is not taken.
- The EXCAUSE field in SEQSTAT is updated with an unrecoverable event code.
- The address of the offending instruction is saved in RETX.

To determine whether an exception occurred while an exception handler was executing, check SEQSTAT at the end of the exception handler. If an exception has occurred, register RETX holds the address of the instruction that caused the exception.

At the end of the exception handler, the return address is used to return normally to lower interrupt level code. Whether an exception occurs during the execution of an NMI, emulation, or reset event, the same set of actions occurs.

# Hardware Error Interrupt

The hardware error interrupt indicates a hardware error or system malfunction. Hardware errors occur when logic external to the core, such as a memory bus controller, is unable to complete a data transfer (read or write) and asserts the core's error input signal. Such hardware errors invoke the hardware error interrupt (interrupt IVHW in the Event Vector Table (EVT) and ILAT, IMASK, and IPEND registers). The hardware error interrupt service routine can then read the cause of the error from the 5-bit HWERRCAUSE field appearing in the Sequencer Status register (SEQSTAT) and respond accordingly.

The hardware error interrupt is generated by:

- Attempted access to a reserved memory location
- Attempted access to uninitialized external memory space
- Bus parity errors
- Internal error conditions within the core, such as Performance Monitor overflow
- The DMA Access Bus Comparator interrupt (attempted write to an active DMA register)
- Peripheral errors
- Bus timeout errors

The list of supported hardware conditions, with their related HWERRCAUSE codes, appears in Table 4-13. The bit code for the most recent error appears in the HWERRCAUSE field. If multiple hardware errors occur simultaneously, only the last one can be recognized and serviced. The core does not support prioritizing, pipelining, or queuing multiple error codes. The hardware error interrupt remains active as long as any of the error conditions remain active.

Table 4-13. Hardware Conditions Causing Hardware Error Interrupts

| **Hardware Condition** | **HWERRCAUSE (5 bits)** | **Notes / Examples** |
|---|---|---|
| DMA Bus Comparator Source | 0b00001 | The Compare Hit output is routed directly to the Hardware Error interrupt input. The Compare Hit interrupt is maskable by writing to the DMA Bus Control Comparator register (DB_CCOMP). See "DMA Bus Debug Registers" on page 20-29. |
| Performance Monitor Overflow | 0b10010 | |

Table 4-13. Hardware Conditions Causing Hardware Error Interrupts

| Hardware Condition | HWERRCAUSE (5 bits) | Notes / Examples |
|---|---|---|
| Error accessing reserved or undefined memory | 0b10110 | |
| Error accessing reserved or undefined memory | 0b10111 | |
| RAISE instruction | 0b11000 | Software issued a RAISE instruction to invoke the hardware error interrupt (IVHW). |
| Reserved | All other bit combinations. | |

## Core Timer

The Core Timer Interrupt (IVTMR) is triggered when the core timer value reaches zero. See "Timers" on page 16-1.

## General-purpose Interrupts (IVG7-IVG15)

General-purpose interrupts are used for any event that requires processor attention. For instance, a DMA controller may use them to signal the end of a data transmission, or a serial communications device may use them to signal transmission errors.

Software can also trigger general-purpose interrupts by using the RAISE instruction. The RAISE instruction can force events for interrupts IVG15-IVG7, IVTMR, IVHW, NMI, and RST, but not for exceptions and emulation (EVX and EMU, respectively).

ⓘ It is recommended to reserve the two lowest priority interrupts (IVG15 and IVG14) for software interrupt handlers.

# Servicing Interrupts

The CEC has a single interrupt queueing element per event, as a bit in the ILAT register. The appropriate ILAT bit is set when an interrupt rising edge is detected (which takes 2 core clock cycles) and cleared when the respective IPEND register bit is set. The IPEND bit indicates that the event vector has entered the core pipeline. At this point, the CEC recognizes and queues the next rising edge event on the corresponding interrupt input. The minimum latency from the rising edge transition of the general-purpose interrupt to the IPEND output asserted is three core clock cycles. However, the latency can be much higher, depending on the core's activity level and state.

To determine when to service an interrupt, the controller logically ANDs the three quantities in ILAT, IMASK, and the current processor priority level.

Servicing the highest priority interrupt involves these actions:

1. The interrupt vector in the EVT becomes the next fetch address.

   On an interrupt, all instructions currently in the pipeline are aborted. On a service exception, all instructions after the excepting instruction are aborted. On an error exception, the excepting instruction and all instructions after it are aborted.

2. The return address is saved in the appropriate return register.

   The return register is RETI for interrupts, RETX for exceptions, RETN for NMIs, and RETE for debug emulation. The return address is the address of the instruction after the last-executed instruction from normal program flow.

3. Processor mode is set to the level of the event taken.

If the event is an NMI, exception, or interrupt, the processor mode is Supervisor. If the event is an emulation exception, the processor mode is Emulation.

4. Before the first instruction starts execution, the corresponding interrupt bit in ILAT is cleared and the corresponding bit in IPEND is set.

   Bit IPEND[4] is also set to disable all interrupts until the return address in RETI is saved.

# Interrupts With and Without Nesting

Interrupts are handled either with or without nesting. If interrupts do not require nesting, all interrupts are disabled during the interrupt service routine. Note, however, that emulation, NMI, and exceptions are still accepted by the system.

When the system does not need to support nested interrupts, there is no need to store the return address held in RETI. Only the portion of the machine state used in the interrupt service routine must be saved in the Supervisor stack. To return from a non-nested interrupt service routine, only the RTI instruction must be executed, because the return address is already held in the RETI register.

Figure 4-15 shows an example of interrupt handling where interrupts are globally disabled for the entire interrupt service routine.

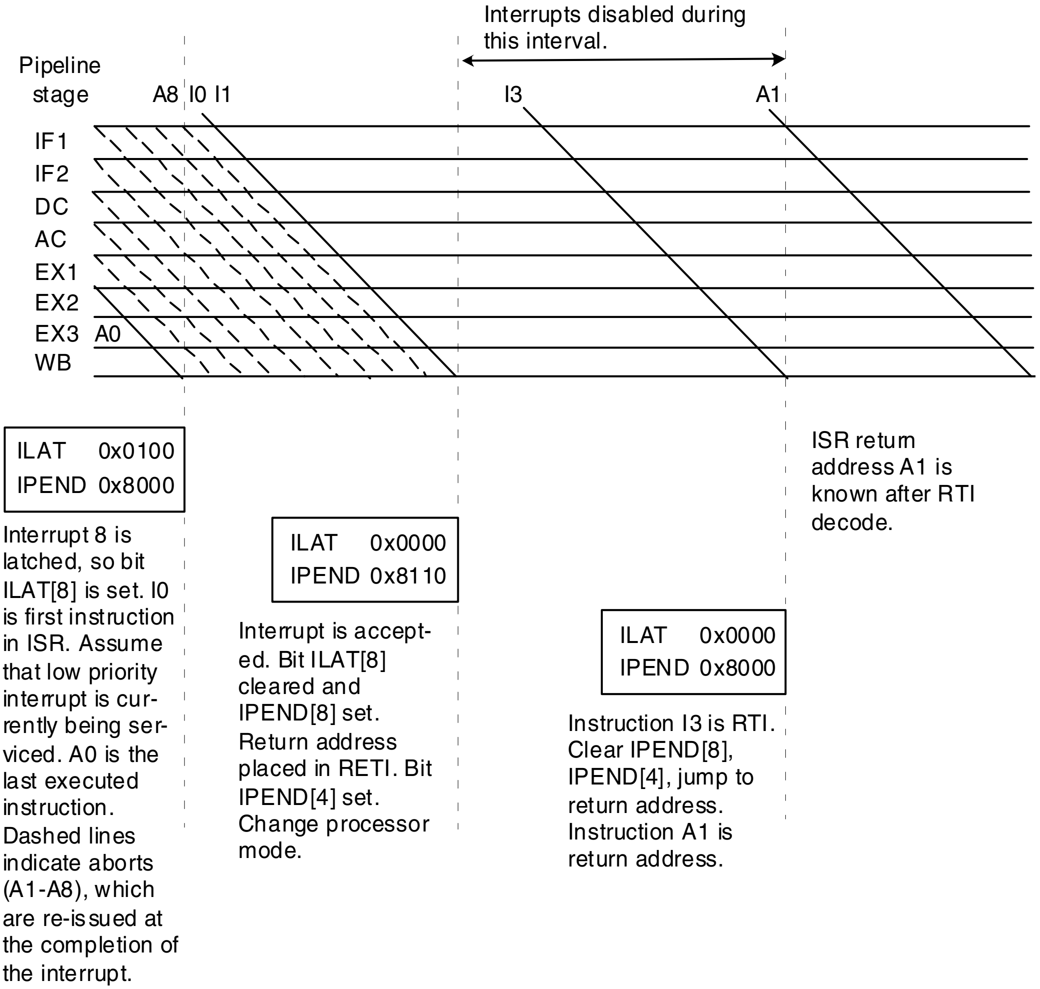

Interrupts disabled during this interval.
Pipeline stage
A8
I0
I1
I3
A1
IF1
IF2
DC
AC
EX1
EX2
EX3
A0
WB
ILAT 0x0100
IPEND 0x8000
Interrupt 8 is latched, so bit ILAT[8] is set. I0 is first instruction in ISR. Assume that low priority interrupt is currently being serviced. A0 is the last executed instruction. Dashed lines indicate aborts (A1-A8), which are re-issued at the completion of the interrupt.
ILAT 0x0000
IPEND 0x8110
Interrupt is accepted. Bit ILAT[8] cleared and IPEND[8] set. Return address placed in RETI. Bit IPEND[4] set. Change processor mode.
ILAT 0x0000
IPEND 0x8000
Instruction I3 is RTI. Clear IPEND[8], IPEND[4], jump to return address. Instruction A1 is return address.
ISR return address A1 is known after RTI decode.

Figure 4-15. Non-Nested Interrupt Handling

If interrupts require nesting, the return address to the interrupted point in the original ISR must be explicitly saved and subsequently restored when execution of the nested ISR has completed. The first instruction in an interrupt service routine that supports nesting must save the return address currently held in RETI by pushing it onto the Supervisor stack ([--SP]=RETI). This clears the global interrupt disable bit IPEND[4], enabling interrupts. Next, all registers that are modified by the interrupt service routine are saved onto the Supervisor stack. Processor state is stored in the Supervisor stack, not in the User stack. Hence, the instructions to push RETI ([--SP]=RETI) and pop RETI (RETI=[SP++]) use the Supervisor stack.

Figure 4-16 illustrates that by pushing RETI onto the stack, interrupts can be re-enabled during an ISR, resulting in only a short duration where interrupts are globally disabled.

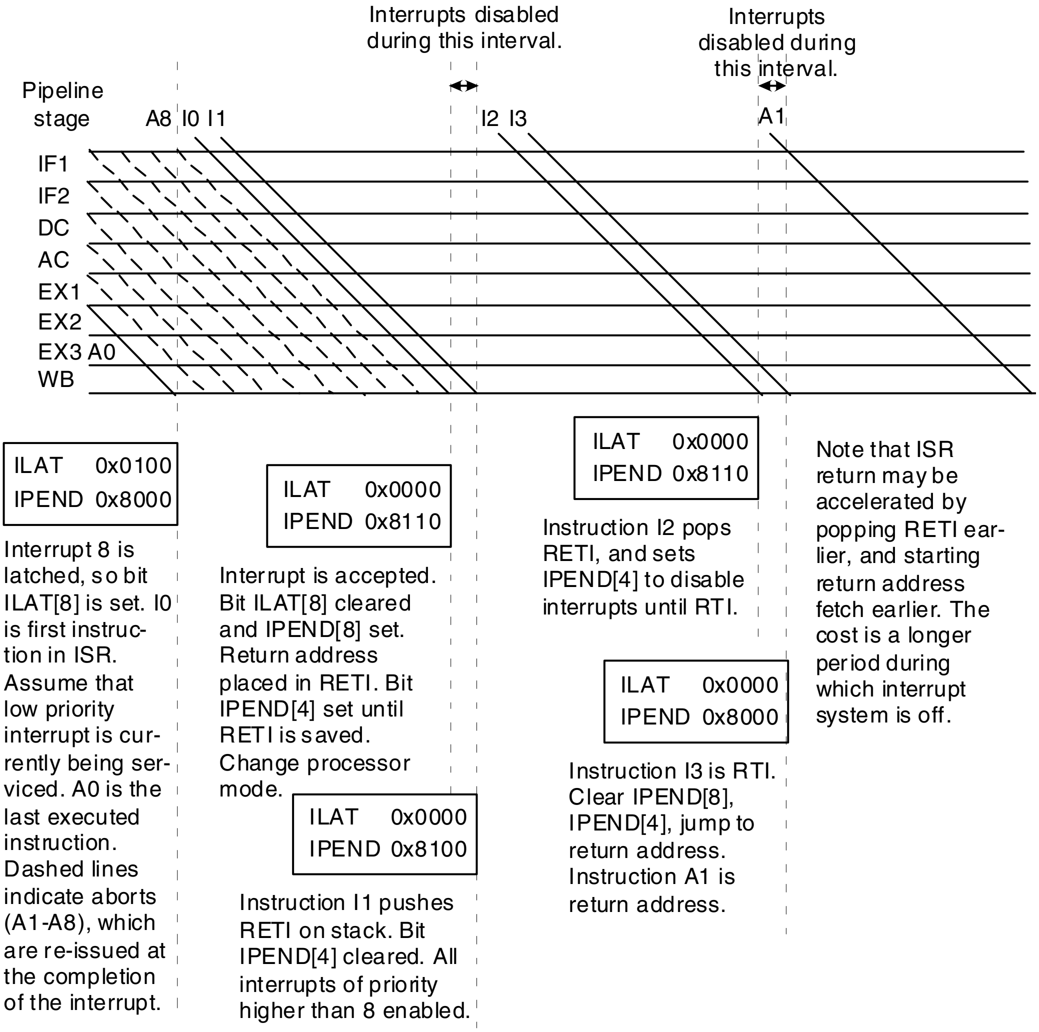

Interrupts disabled during this interval.
Interrupts disabled during this interval.
Pipeline stage
A8
I0
I1
I2
I3
A1
IF1
IF2
DC
AC
EX1
EX2
EX3
A0
WB
ILAT 0x0100
IPEND 0x8000
Interrupt 8 is latched, so bit ILAT[8] is set. I0 is first instruction in ISR. Assume that low priority interrupt is currently being serviced. A0 is the last executed instruction. Dashed lines indicate aborts (A1-A8), which are re-issued at the completion of the interrupt.
ILAT 0x0000
IPEND 0x8110
Interrupt is accepted. Bit ILAT[8] cleared and IPEND[8] set. Return address placed in RETI. Bit IPEND[4] set until RETI is saved. Change processor mode.
ILAT 0x0000
IPEND 0x8100
Instruction I1 pushes RETI on stack. Bit IPEND[4] cleared. All interrupts of priority higher than 8 enabled.
ILAT 0x0000
IPEND 0x8110
Instruction I2 pops RETI, and sets IPEND[4] to disable interrupts until RTI.
ILAT 0x0000
IPEND 0x8000
Instruction I3 is RTI. Clear IPEND[8], IPEND[4], jump to return address. Instruction A1 is return address.
Note that ISR return may be accelerated by popping RETI earlier, and starting return address fetch earlier. The cost is a longer period during which interrupt system is off.

Figure 4-16. Nested Interrupt Handling

## Example Prolog Code for Nested Interrupt Service Routine

Listing 4-2. Prolog Code for Nested ISR

```
/* Prolog code for nested interrupt service routine. Push return
address in RETI into Supervisor stack, ensuring that interrupts
are back on. Until now, interrupts have been suspended. */
ISR:
[--SP] = RETI ; /* Enables interrupts and saves return address to
stack */
[--SP] = ASTAT ;
[--SP] = FP ;
[-- SP] = (R7:0, P5:0) ;
/* Body of service routine. Note that none of the DSP resources
(accumulators, DAGs, loop counters and bounds) have been saved.
It's assumed that this interrupt service routine does not use
them. */
```

## Example Epilog Code for Nested Interrupt Service Routine

Listing 4-3. Epilog Code for Nested ISR

```
/* Epilog code for nested-interrupt service routine. */
/* Restore ASTAT, Data, and Pointer registers. Popping RETI from
Supervisor stack ensures that interrupts are suspended between
load of return address and RTI. */
(R7:0, P5:0) = [SP++] ;
FP    = [SP++] ;
ASTAT = [SP++] ;
RETI  = [SP++] ;
```

```
/* Execute RTI, which jumps to return address, re-enables inter-
rupts, and switches to User mode if this is the last nested
interrupt in service. */
RTI;
```

The RTI instruction causes the return from an interrupt. The return address is popped into the RETI register from the stack, an action that suspends interrupts from the time that RETI is restored until RTI finishes executing. The suspension of interrupts prevents a subsequent interrupt from corrupting the RETI register.

Next, the RTI instruction clears the highest priority bit that is currently set in IPEND. The processor then jumps to the address pointed to by the value in the RETI register and re-enables the interrupts by clearing IPEND[4].

## Logging of Nested Interrupt Requests

The SIC detects level-sensitive interrupt requests from the peripherals. The CEC provides edge-sensitive detection for its general-purpose interrupts (IVG7-IVG15). Consequently, the SIC generates a synchronous interrupt pulse to the CEC and then waits for interrupt acknowledgement from the CEC. When the interrupt has been acknowledged by the core (via assertion of the appropriate IPEND output), the SIC generates another synchronous interrupt pulse to the CEC if the peripheral interrupt is still asserted. This way, the system does not lose peripheral interrupt requests that occur during servicing of another interrupt.

Because multiple interrupt sources can map to a single core processor general-purpose interrupt, multiple pulse assertions from the SIC can occur simultaneously, before, or during interrupt processing for an interrupt event that is already detected on this interrupt input. For a shared interrupt, the IPEND interrupt acknowledge mechanism described above re-enables all shared interrupts. If any of the shared interrupt sources are still asserted, at least one pulse is again generated by the SIC. The Interrupt Status registers indicate the current state of the shared interrupt sources.

## Self-Nesting Mode

The nesting method described in the previous section allows an interrupt of higher priority to preempt interrupts of lower priority. However, it does not allow one interrupt to preempt another interrupt of the same priority, nor does it allow nesting at the same priority level.

When self-nested interrupts are not enabled, general-purpose interrupts can only preempt general-purpose interrupts of a lower priority (higher index). For example, when processing interrupt 7, another interrupt 7 request remains pending (ILAT[7] = 1), but the interrupt is not serviced until the current interrupt 7 service routine executes an RTI.

On the other hand, when self-nesting interrupts are enabled, an event interrupts processing at the same interrupt service level, provided RETI is pushed to the stack and interrupts are enabled. For example, when processing interrupt 7 (IPEND[7:0] = 0x80), another interrupt 7 request causes software to again vector to the interrupt 7 service routine.

ⓘ Self-nesting of interrupts applies only to core interrupts or interrupts generated using the RAISE instruction; it is not allowed with peripheral interrupts.

For the interrupt system to allow self-nesting, software must set bit SNEN (self-nesting enable) in SYSCFG. See Figure 4-3 on page 4-7. In self-nesting mode, the system sets the LSB of the address in the return register RETI (RETI[0]) to indicate an incoming interrupt has the same priority as an interrupt that is currently being serviced. The bit RETI[0] is automatically set on entry into an interrupt service routine, and it is simply used as a status bit to flag the processor that the current ISR is self-nesting.

The return-from-interrupt instruction, RTI, is sensitive to the state of RETI[0] and SNEN. When both RETI[0] and SNEN are set, RTI clears only the global disable bit IPEND[4]. However, when self-nesting mode is disabled (SNEN = 0), RTI clears both IPEND[4] and the IPEND bit that corresponds to the current interrupt level.

The SNEN bit should be set once in the reset service routine and not changed again during normal interrupt processing.

Since the LSB of the RETI register is used to store the self-nesting state, avoid changing the contents of the RETI register when self-nesting is enabled, except for saving and restoring the register to the stack.

## Exception Handling

Interrupts and exceptions treat instructions in the pipeline differently:

- When an interrupt occurs, all instructions in the pipeline are aborted.
- When an exception occurs, all instructions in the pipeline after the excepting instruction are aborted. For service exceptions, the excepting instruction is also aborted.

Because exceptions, NMIs, and emulation events have a dedicated return register, guarding the return address is optional. Consequently, the push and pop instructions for exceptions, NMIs, and emulation events do not affect the interrupt system.

Note, however, the return instructions for exceptions (RTX, RTN, and RTE) do clear the least significant bit currently set in IPEND.

## Deferring Exception Processing

Exception handlers are usually long routines, because they must discriminate among several exception causes and take corrective action accordingly. The length of the routines may result in long periods during which the interrupt system is, in effect, suspended.

To avoid lengthy suspension of interrupts, write the exception handler to identify the exception cause, but defer the processing to a low priority interrupt. To set up the low priority interrupt handler, use the Force Interrupt / Reset instruction (`RAISE`).

> When deferring the processing of an exception to lower priority interrupt `IVGx`, the system must guarantee that `IVGx` is entered before returning to the application-level code that issued the exception. If a pending interrupt of higher priority than `IVGx` occurs, it is acceptable to enter the high priority interrupt before `IVGx`.

## Example Code for an Exception Handler

Listing 4-4. Exception Routine Handler with Deferred Processing

```
/* Determine exception cause by examining EXCAUSE field in
SEQSTAT (first save contents of R0, P0, P1 and ASTAT in Supervi-
sor SP) */
[--SP] = R0 ;
[--SP] = P0 ;
[--SP] = P1 ;
[--SP] = ASTAT ;
R0 = SEQSTAT ;
/* Mask the contents of SEQSTAT, and leave only EXCAUSE in R0 */
R0 <<= 26 ;
R0 >>= 26 ;
/* Using jump table EVTABLE, jump to the event pointed by R0 */
P0 = R0 ;
```

```
P1 = _EVTABLE ;
P0 = P1 + ( P0 << 1 ) ;
R0 = W [ P0 ] (Z) ;
P1 = R0 ;
JUMP (PC + P1) ;
/* The entry point for an event is as follows. Here, processing
is deferred to low-priority interrupt IVG15. Also, parame-
ter-passing would typically be done here. */
_EVENT1:
RAISE 15 ;
JUMP.S _EXIT ;
/* Entry for event at IVG14 */
_EVENT2:
RAISE 14 ;
JUMP.S _EXIT ;
/* comments for other events */
/* At the end of handler, restore R0, P0, P1 and ASTAT, and
return. */
_EXIT:
ASTAT = [SP++] ;
P1 = [SP++] ;
P0 = [SP++] ;
R0 = [SP++] ;
RTX ;
_EVTABLE:
.byte2 addr_event1;
.byte2 addr_event2;
...
.byte2 addr_eventN;
/* The jump table EVTABLE holds 16-bit address offsets for each
event. With offsets, this code is position-independent and the
table is small.
+--------------+
| addr_event1  | _EVTABLE
```

```
+--------------+
| addr_event2  | _EVTABLE + 2
+--------------+
|     . . .    |
+--------------+
| addr_eventN  | _EVTABLE + 2N
+--------------+
*/
```

## Example Code for an Exception Routine

Listing 4-5 provides an example framework for an exception routine that would be jumped to from an exception handler such as that described above.

Listing 4-5. Exception Routine

```
[--SP] = RETX ;   /* Push return address on stack. No change to
ILAT or IPEND. */

/* Put body of exception routine here. */

RETX = [SP++] ;   /* To return, pop return address and jump. No
change to ILAT or IPEND. */

RTX ;   /* Return from exception. Clear IPEND[3] (Exception Pend-
ing bit). */
```

## Executing RTX, RTN, or RTE in a Lower Priority Event

Instructions RTX, RTN, and RTE are designed to return from an exception, NMI, or emulator event, respectively. Do not use them to return from a lower-priority event. To return from an interrupt, use the RTI instruction. Failure to use the correct instruction produces the following results.

- If a program mistakenly uses RTX, RTN, or RTE to return from an interrupt, the core branches to the address in the corresponding return register (RETX, RETN, RETE) and leaves IPEND unaffected.
- If a program mistakenly uses RTI or RTX to return from an NMI routine, the core branches to the address in the corresponding return register (RETI, RETX), and clears the bit in IPEND that corresponds to the return instruction.

  In the case of RTX, bit IPEND[3] is cleared. In the case of RTI, the bit of the highest priority interrupt in IPEND is cleared.

## Recommendation for Allocating the System Stack

The software stack model for processing exceptions implies that the Supervisor stack must never generate an exception while the exception handler is saving its state. However, if the Supervisor stack grows past a CPLB entry or SRAM block, it may, in fact, generate an exception.

To guarantee that the Supervisor stack never generates an exception—never overflows past a CPLB entry or SRAM block while executing the exception handler—calculate the maximum space that all interrupt service routines and the exception handler occupy while they are active, and then allocate this amount of SRAM memory.

## Latency in Servicing Events

In some DSP architectures, if instructions are executed from external memory and an interrupt occurs while the instruction fetch operation is underway, then the interrupt is held off from being serviced until the current fetch operation has completed. Consider a processor operating at 300 MHz and executing code from external memory with 100 ns access times. Depending on when the interrupt occurs in the instruction fetch operation, the interrupt service routine (ISR) may be held off for around 30 instruction clock cycles. When cache line fill operations are taken into account, the ISR could be held off for many hundreds of cycles.

In order for high-priority interrupts to be serviced with the least latency possible, the ADSP-21535 processor allows any high latency fill operation to be completed at the system level, while an ISR executes from L1 memory. Figure 4-17 illustrates this concept.

If an ADSP-21535 DSP instruction load operation misses the L1 Instruction Cache and generates a high latency line fill operation on the SBIU, then when an interrupt occurs, it is not held off until the fill has completed. Instead, the processor executes the ISR in its new context, and the cache fill operation completes in the background.

Note the ISR must reside in L1 cache or SRAM memory and must not generate a cache miss, an L2 memory access, or a peripheral access, as the SBIU is already busy completing the original cache line fill operation. If a load or store operation is executed in the ISR requiring the SBIU, then the ISR is held off while the original external access is completed, before initiating the new load or store.

If the ISR finishes execution before the load operation has completed, then the ADSP-21535 processor continues to stall, waiting for the fill to complete.

This same behavior is also exhibited for stalls involving reads of slow data memory or peripherals.

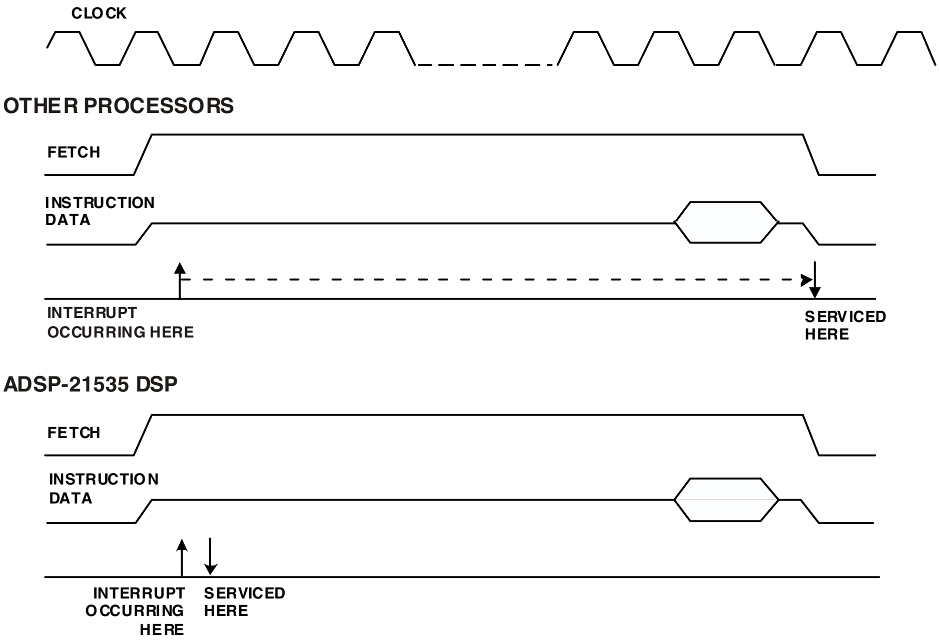


Figure 4-17. Minimizing Latency in Servicing an ISR

Writes to slow memory generally do not show this behavior, as the writes are deemed to be single cycle, being immediately transferred to the write buffer for subsequent execution.

For detailed information about cache and memory structures, see "Memory" on page 6-1.