

8 PROGRAM SEQUENCER

The ADSP-TS201 TigerSHARC processor core contains a program sequencer. The sequencer contains the instruction alignment buffer (IAB), program counter (PC), branch target buffer (BTB), interrupt manager, and address fetch mechanism. Using these features and the instruction pipeline, the sequencer (highlighted in [Figure 8-1](#)) manages program execution.

The sequencer fetches instructions from memory and executes program flow control instructions. The operations that the sequencer supports include:

- Supply address of next instruction to fetch
- Maintain instruction alignment buffer (IAB) by buffering fetched instructions
- Maintain branch target buffer (BTB) by reducing branch delays
- Decrement loop counters
- Evaluate conditions (for conditional instructions)
- Respond to interrupts (with changes to program flow)

Figure 8-2 shows a detailed block diagram of the sequencer. Looking at this diagram, note the following blocks within the sequencer.

- The program counter (PC) increments the fetch address for linear flow or modifies the fetch address as needed for non-linear flow (branches, loops, interrupts, or others).
- The branch target buffer caches addresses for branches to reduce pipeline costs on predicted branches. For more information, see [“Branching Execution” on page 8-16](#).
- The fetch unit puts the address on the bus for the next quad word to fetch from memory.
- The instruction alignment buffer receives the instruction quad words from memory, buffers them, and distributes the instructions to the compute blocks, IALUs, and program sequencer.

With the functional blocks shown in [Figure 8-2](#), the sequencer can support a number of program flow variations. Program flow in the processor is mostly linear with the processor executing program instructions sequentially. This linear flow varies occasionally when the program uses non-sequential program structures, such as those illustrated in [Figure 8-3](#). Non-sequential structures direct the processor to execute an instruction that is not at the next sequential address. These structures include:

- **Loops.** One sequence of instructions executes several times with near-zero overhead.
- **Subroutines.** The processor temporarily redirects sequential flow to execute instructions from another part of program memory.
- **Jumps.** Program flow transfers permanently to another part of program memory.

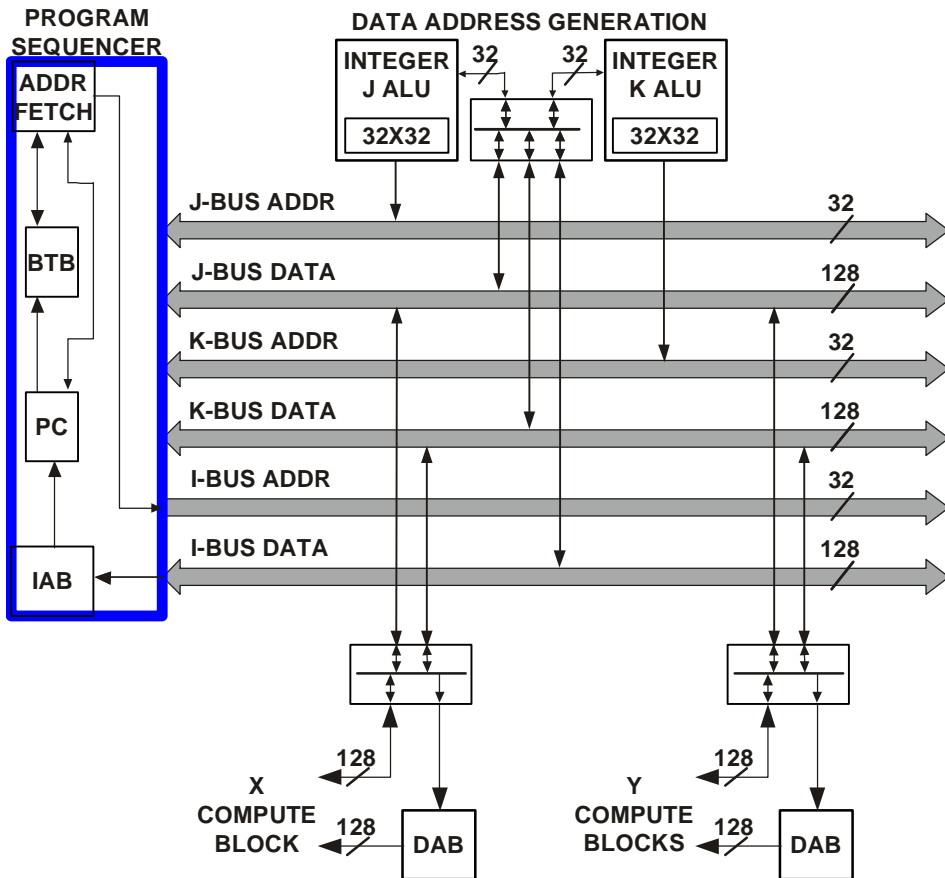


Figure 8-1. Program Sequencer

- **Interrupts.** Subroutines in which a runtime event triggers the execution of the routine.
- **Idle.** An instruction that causes the processor to cease operations, holding its current state until an interrupt occurs. Then, the processor services the interrupt and continues normal execution.

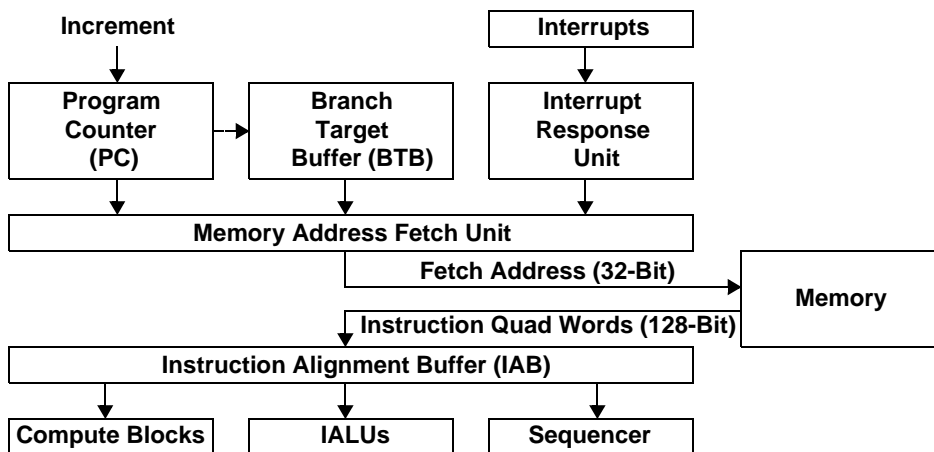


Figure 8-2. Sequencer Detailed Block Diagram

For information on using each type of program flow, see [“Sequencer Operations” on page 8-8](#).

To support optimized flow of execution, the ADSP-TS201 processor uses an instruction pipeline. The processor fetches quad words from memory, parses the quad words into instruction lines (consisting of one to four instructions), decodes the instructions, and executes them. [Figure 8-4](#) shows the instruction pipeline stages and shows how the pipeline interacts with the BTB, IAB, and memory pipeline.

From start to finish, an instruction line requires at least ten cycles to traverse the pipeline. The execution throughput is one instruction line every processor core clock (CCLK) cycle. Due to the execution of IALU instructions at the Integer (I) stage, execution of compute block instructions at the Execute 2 (EX2) stage, and memory access effects, the full flow cannot be analyzed as a single ten-stage pipeline, but must be viewed as sequential unit pipes.

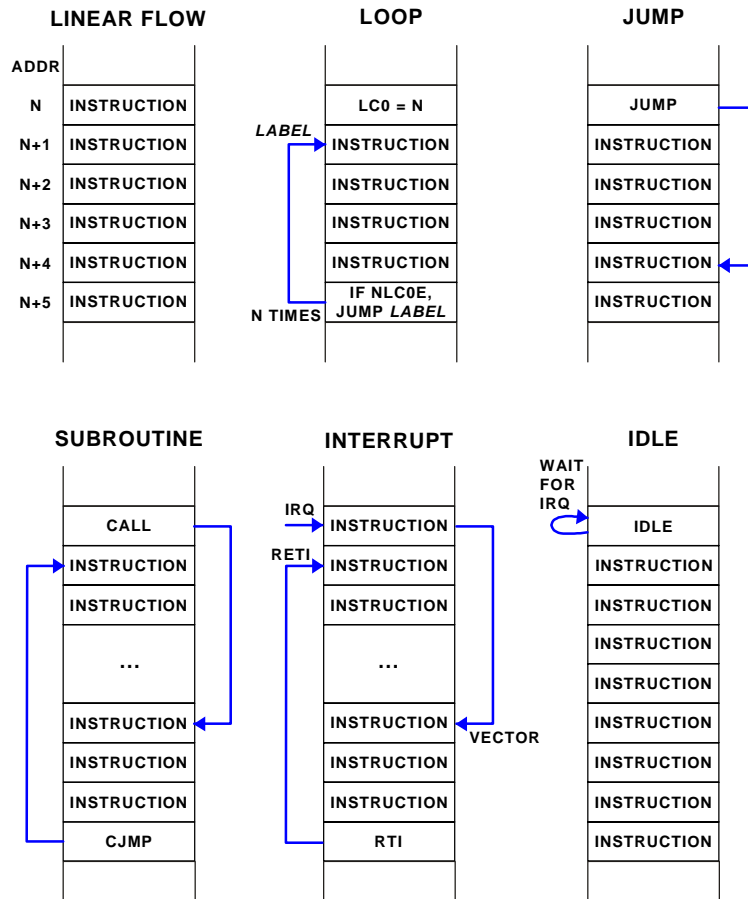


Figure 8-3. Program Flow Variations

The instruction fetch (F1, F2, F3, and F4) pipe stages are common to all instructions and are driven by memory accesses. The F1 stage interacts with the BTB to minimize overhead cycles when branching. For more information, see [“Branching Execution” on page 8-16](#) and [“Branch Target Buffer \(BTB\)” on page 8-38](#).

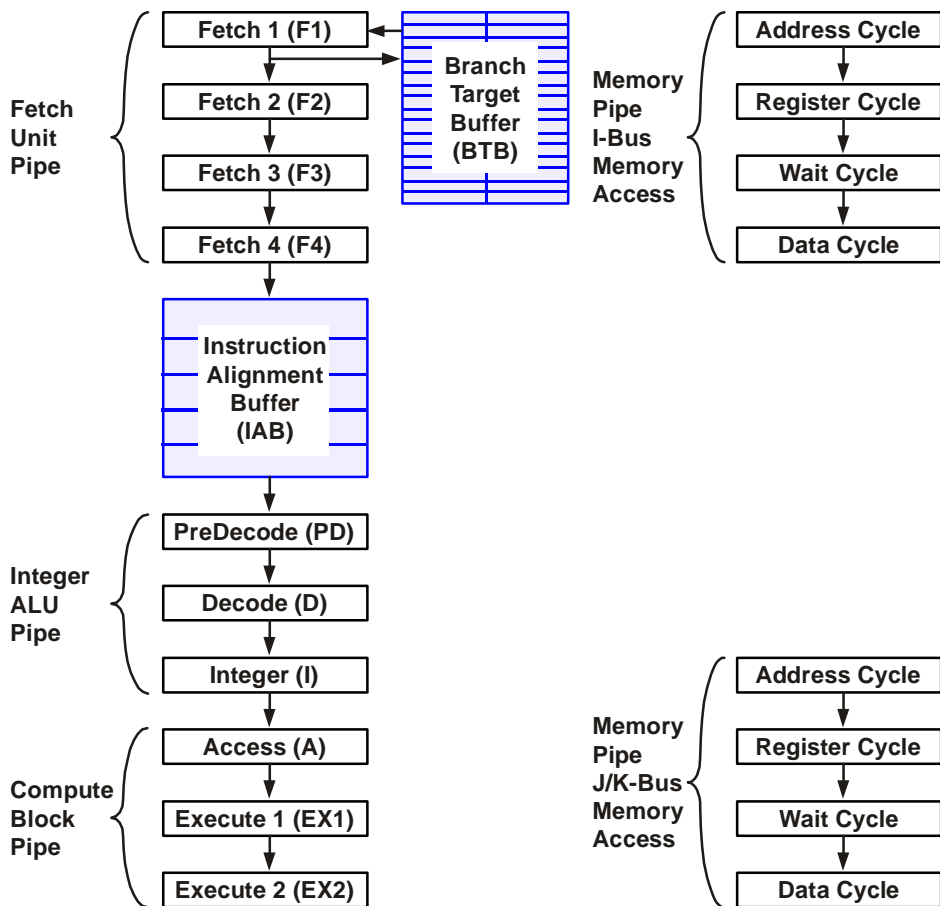


Figure 8-4. Instruction Pipeline, IAB, and BTB Versus Memory Pipeline

The remaining pipe stages are instruction driven. The execution differs between the IALU, compute block, and sequencer. The instruction driven pipe stages are PreDecode (PD), Decode (D), Integer (I), operand Access (A), Execute 1 (EX1), and Execute 2 (EX2).

In the instruction driven pipeline stages, instruction pipe details differ according to the unit executing the instruction:

PreDecode	The sequencer extracts the next instruction line, uses the IAB to distribute it to the respective execution units (compute blocks, IALUs, and sequencer), and updates the program counter. For more information, see “Instruction Alignment Buffer (IAB)” on page 8-34 .
Decode	The IALUs decode instructions—The compute blocks transfer instructions to the computation units (ALU, multiplier, and shifter).
Integer	The IALUs execute arithmetic instruction, return results, and update flags—The computational units decode instruction and check for dependencies.
Access	The IALUs begin memory access (for applicable instructions)—The computational units select source registers in the register files.
Execute 1 and 2	The IALUs complete execution of memory access instruction—The computational units complete execution, return results, and update flags. Note: The ADSP-TS101 processor uses EX1, while the ADSP-TS201 processor uses EX2.

These differences in pipe operation between IALUs and computational units cause different pipeline effects when branching. For more information, see [“Instruction Pipeline Operations” on page 8-30](#).




As shown in [Figure 8-4](#), the memory pipeline operates in parallel with the instruction pipeline. Stalls in the Address Cycle or Register Cycle of the memory pipeline appear as stalls in the corresponding stages of the instruction pipeline. For more information, see [“Dependency and Resource Effects on Pipeline” on page 8-59](#) and the *ADSP-TS201 TigerSHARC Processor Hardware Reference*.

Sequencer Operations

Sequencer operations support linear execution (one sequential address after another) and non-linear execution (transferring execution to a non-sequential address). These operations are described in these sections:

- [“Conditional Execution” on page 8-12](#)
- [“Branching Execution” on page 8-16](#)
- [“Looping Execution” on page 8-19](#)
- [“Interrupting Execution” on page 8-20](#)

 Depending on programming and pipeline effects, some branching variations require the processor to automatically insert stall cycles. For more information on pipeline effects and stalls, see [“Instruction Pipeline Operations” on page 8-30](#).

Conditional instructions begin with the syntax *IF Condition*.

The *Condition* can be:

- any status flag from a compute block or IALU operation (for example, AZ, JZ, MV, and others)
- any static flag from the SFREG register
- negated status/static flag (prefixed with N)

Almost all processor instructions can be conditional. The exceptions are a small set of sequencer instructions that may not be conditional. The always unconditional instructions are NOP, IDLE, BTBINV, TRAP, and EMUTRAP.

In the sequencer, there are two registers that provide control and status. The sequencer control (SQCTL) and sequencer status (SQSTAT) registers appear in [Figure 8-5](#), [Figure 8-7](#), [Figure 8-6](#), and [Figure 8-8](#). These registers support:

- **Normal mode.** Using the normal mode (NMOD) bit, programs select user mode (=0) in which programs can only access the compute block and IALU registers or supervisor mode (=1) in which programs have unlimited register access. After booting and when responding to an interrupt, the processor automatically goes into supervisor mode. There is a three-cycle latency on explicitly switching between these modes.
- **Software reset.** Using the software reset (SWRST) bit, programs can reset the processor core.
- **Global interrupt enable.** Using the global interrupt enable (GIE) bit, programs can globally mask interrupts.
- **Exception mask/pointer.** Using the exception (EXCEPT) bits in the SQCTL and SQSTAT registers, programs can mask, unmask, and nest software interrupts (exceptions).
- **Flag status.** Using the flag input (FLGX) bits, programs can observe the input value of an input flag pin.¹ Input/output control for flag pins is available in the FLAGREG register.
- **Branch target buffer control.** Using the BTB enable (BTBEN) and BTB lock (BTBLK) bits, programs control the BTB operation.



Other status information is also available. See [Figure 8-5](#), [Figure 8-7](#), [Figure 8-6](#), and [Figure 8-8](#).

¹ The flag pin bit updates the corresponding flag pin after a delay of between 1 and 3 SCLK cycles. Setting and clearing a flag bit might not affect the pin if both operations occur during that delay. The recommended way to set and clear the flag pin bit is to get an external indication that the bit has been set before clearing it. Otherwise, insert (2 x LCLKRAT) instruction lines between the set and the clear to compensate for the delay.

Sequencer Operations

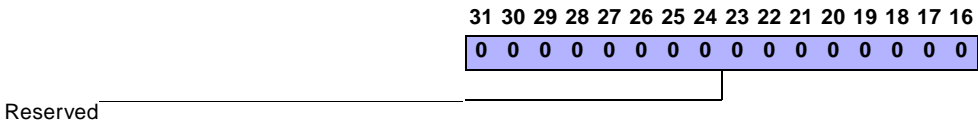


Figure 8-5. SQCTL (Upper) Register Bit Descriptions

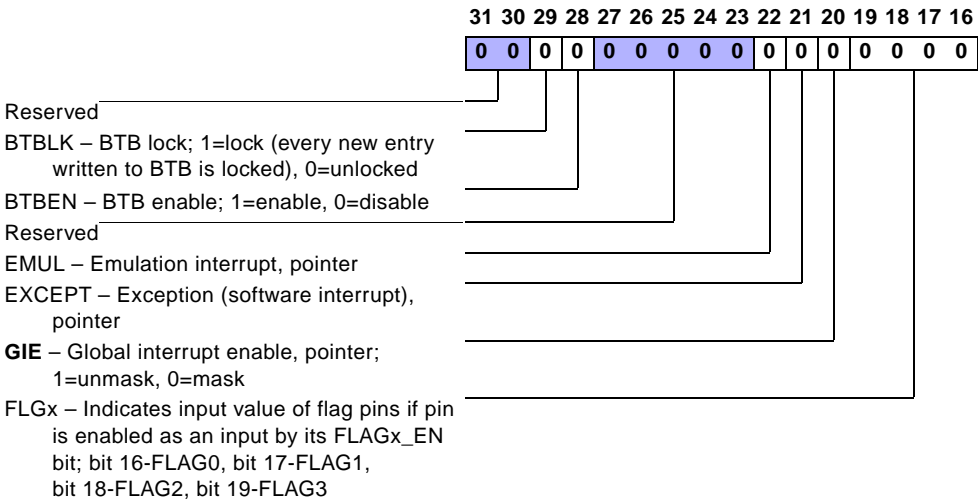


Figure 8-6. SQSTAT (Upper) Register Bit Descriptions

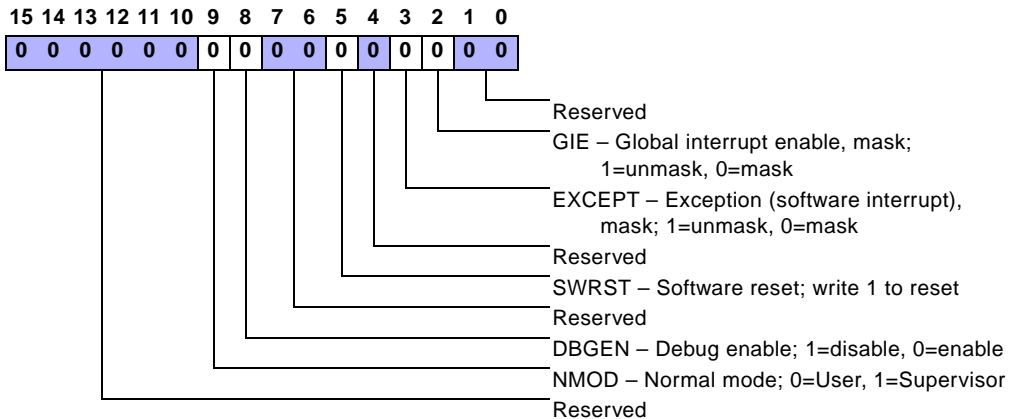


Figure 8-7. SQCTL (Lower) Register Bit Descriptions

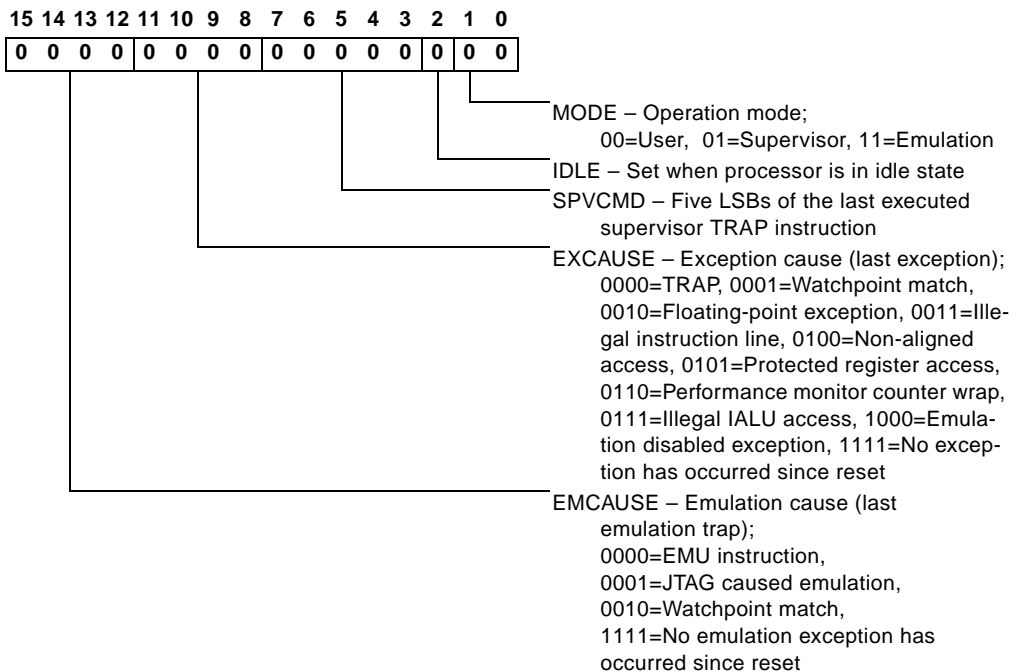


Figure 8-8. SQSTAT (Lower) Register Bit Descriptions

Conditional Execution

All ADSP-TS201 processor instructions¹ can be executed conditionally (a mechanism also known as predicated execution). The condition field exists in one instruction in an instruction line, and all the remaining instructions in that line either execute or not—depending on the outcome of the condition—or execute unconditionally.

In a conditional computational instruction, the execution of the entire instruction line can depend on the specified condition at the beginning of the instruction line. Conditional computational instructions take the form:

```
IF Condition;  
    D0, Instruction; D0, Instruction; D0, Instruction ;;
```

This syntax permits up to three instructions to be controlled by a condition. Omitting the D0 before the instruction makes the instruction unconditional.

[Listing 8-1](#) shows some example conditional ALU instructions. For a description of the ALU conditions used in these examples, see [“ALU Execution Conditions” on page 3-13](#).

Listing 8-1. Conditional Compute and IALU Instructions

```
IF XALT; D0, R3 = R1 + R2 ;;  
/* conditional execution of the add in compute blocks X and Y is  
based on the ALT condition in compute block X */  
  
IF YAEQ; D0, XR0 = R1 + R2 ;;  
/* conditional execution of the add in compute block X is based  
on the AEQ condition in compute block Y */
```

¹ Except for NOP, IDLE, BTBINV, TRAP, and EMUTRAP

```
IF ALE; D0, R0 = R1 + R2 ;;  
/* conditional execution of the add in compute blocks X and Y is  
based on the ALE condition in compute block X for execution in X  
and is based on the ALE condition in compute block Y for execu-  
tion in Y */
```

```
IF ALE; D0, XR0 = R1 + R2 ;;  
/* conditional execution of the add in compute block X is based  
on the ALE condition in compute block X */
```

```
IF ALE; D0, R0 = [J0 + J1] ;;  
/* conditional execution of load is based on ORing the ALE condi-  
tion in compute blocks X and Y */
```

In a conditional program sequencer instruction that is based on an ALU condition, the execution of the program sequencer instruction line depends on the specified ALU condition at the beginning of the instruction line. These conditional program sequencer instructions take the form:

```
IF Condition, Sequencer_Instruction;  
ELSE, Instruction; ELSE, Instruction; ELSE, Instruction ;;
```

This syntax permits up to three instructions to be controlled by a condition. Omitting the ELSE before the instruction makes the instruction unconditional.

This syntax permits program sequencer instructions to be based on computational conditions. [Listing 8-2](#) shows some example conditional program sequencer instructions based on ALU conditions.

Sequencer Operations

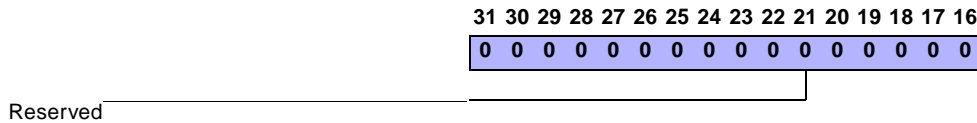


Figure 8-9. SFREG (Upper) Register Bit Descriptions

Listing 8-2. Conditional Sequencer Instructions

```
IF ALE, JUMP label;;
/* conditional execution of the jump is based on ORing the ALE
condition in compute blocks X and Y */

IF XALE, JUMP label;;
/* conditional execution of the jump is based on ALE condition in
compute block X */

IF AEQ, JUMP label; ELSE, XR0 = R5 + R6; YR8 = R9 - R10;;
/* conditional execution of the jump is based on ORing the AEQ
condition in compute blocks X and Y; the add in compute block X
only executes if the jump does not execute; the subtract in com-
pute block Y is unconditional (always executes) */
```

Besides the requirement that any sequencer instruction must use the first instruction slot (each instruction line contains four slots), it is important to note that the address used in a sequencer branch instruction (for example, `JUMP Address`) determines whether the instruction uses one slot or two. If a sequencer instruction specifies a relative or absolute address greater than 16 bits, the processor automatically uses an immediate extension (the second instruction slot) to hold the extra address bits.

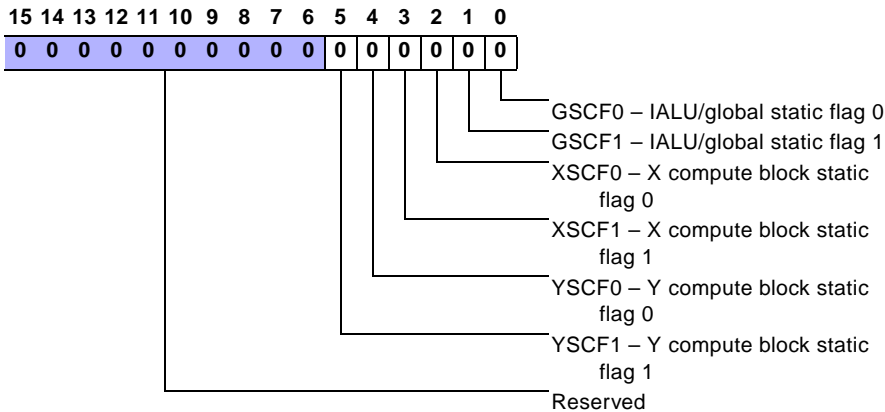


Figure 8-10. SFREG (Lower) Register Bit Descriptions

When a program *Label* is used instead of an address, the assembler converts the *Label* to a PC-relative address. When the *Label* is contained in the same program `.SECTION` as the branch instruction, the assembler uses a 16-bit address. When the *Label* is *not* contained in the same program `.SECTION` as the branch instruction, the assembler uses a 32-bit address. For more information on PC-relative and absolute addresses and branching, see [“Branching Execution” on page 8-16](#).

To provide conditional instruction support for static conditions, the sequencer has a static flag (SFREG) register (appears in [Figure 8-9](#) and [Figure 8-10](#)). The sequencer uses this register to store status flag values from the compute blocks and IALUs for later usage in conditional instructions.

For examples using the SFREG conditions in the compute blocks and IALUs, see [“ALU Static Flags” on page 3-14](#), [“Multiplier Static Flags” on page 5-21](#), [“Shifter Static Flags” on page 6-17](#), [“ALU Static Flags” on page 3-14](#), and [“IALU Static Flags” on page 7-14](#).

Sequencer Operations

i The SFREG register has a three cycle load latency; values loaded to this register are not available for usage for three cycles after the register load. If the SFREG register is used in a conditional instruction during the latency cycles, there is no data dependency stall and the previous SFREG value is used.

Branching Execution

The sequencer supports branching execution with the JUMP, CALL, CJMP_CALL, CJMP, RETI, RDS, and RTI instructions. [Figure 8-3 on page 8-5](#) provides a high-level comparison between all branch variations. Looking at [Figure 8-11](#), note some additional details about the JUMP, CALL, and CJMP_CALL operations.

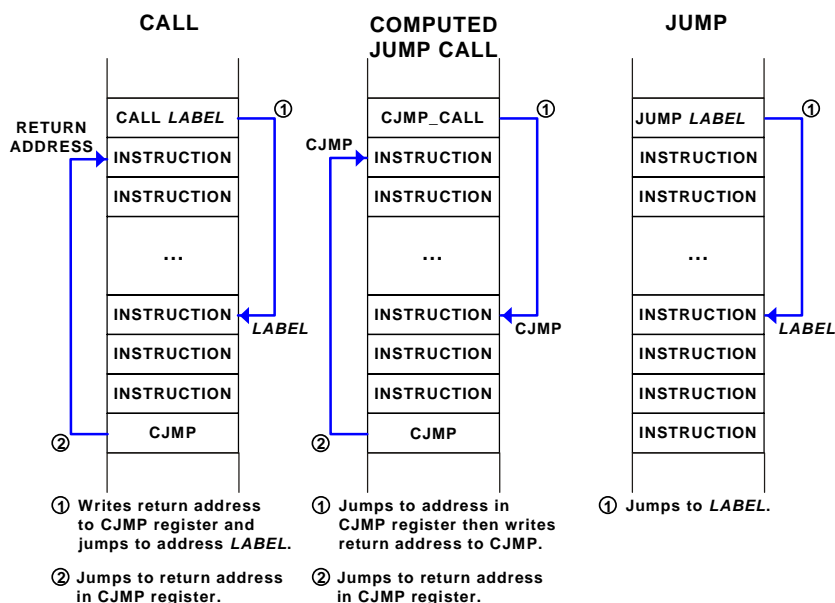


Figure 8-11. Call Versus Computed Jump Call Versus Jump

A `CALL` instruction transfers execution to address *Label* (or to an immediate 16- or 32-bit address). When processing a call, the sequencer writes the return address (next sequential address after the call) to the `CJMP` register, then jumps to the subroutine address. The `CJMP` instruction at the end of the subroutine causes the sequencer to jump to the address in `CJMP`.

A `CJMP_CALL` instruction transfers execution to a subroutine using a computed jump address (`CJMP` register). One way to load the computed jump address is to use the (`CJMP`) option on the IALU add/subtract instruction. The `CJMP_CALL` transfers execution to the address indicated by the `CJMP` register, then loads the return address into `CJMP`. The `CJMP` instruction at the end of the subroutine causes the sequencer to jump to the address in `CJMP`.

A `JUMP` instruction transfers execution to address *Label* (or an immediate 16- or 32-bit address).

Branching sequencer instructions use PC-relative or absolute addresses. For relative addressing, the program sequencer bases the address for relative branch on the 32-bit Program Counter. For absolute addressing instructions, the branch instruction provides an immediate 32-bit address value. The PC allows linear addressing of the full 32-bit address range.



If a Program Counter's 32-bit relative or absolute address is used with a `CALL` or `JUMP` instruction (or if the *Label* is located outside the program `.SECTION` containing the `CALL` or `JUMP` instruction), the `CALL` or `JUMP` instruction requires the immediate extension instruction slot to hold the 16-bit address extension.

Sequencer Operations

The default operation of `CALL` and `JUMP` instructions is to assume the address in the instruction is *PC-relative*. To use an *absolute address*, branch instructions use the absolute (ABS) option. The following example shows a PC-relative address branch instruction and an absolute address branch instruction.

```
JUMP fft_routine ;;
/* fft_routine is a program label that appears within this .section; the assembler converts the label into a 16-bit PC-relative address */
CALL iir_routine (ABS) ;;
/* iir_routine is a program label from outside this .section; the assembler converts the label into an absolute address (because the ABS option is used); also the label converts to a 32-bit address (because it is outside this section) */
```

For more examples of branch instructions, see [“Sequencer Examples” on page 8-81](#).

Another default operation that applies to all conditional branch instructions is that the sequencer assumes the conditional test is `TRUE` and predicts the branch is taken—a *predicted branch*. When for programming reasons the programmer can predict that most of the time the branch is not taken, the branch is called not predicted and is indicated as such using the not predicted (NP) option. In the following example, the first conditional branch is a predicted branch and the second is a *not predicted branch*:

```
IF AEQ, JUMP fft_routine ;;
/* this conditional branching instruction is a predicted branch */
IF MEQ, CALL iir_routine (ABS) (NP) ;;
/* this conditional branching instruction is a not predicted branch */
```



Unconditional branches are treated as conditional branches by the sequencer. The sequencer treats unconditional branches as though they are prefixed with the condition `IF TRUE`.

Correctly predicting a branch as taken or not taken is extremely important for pipeline performance. The sequencer writes information about every predicted branch into BTB. The BTB examines the flow of addresses during the pipeline stage Fetch 1.

When a BTB hit occurs (the BTB recognizes the address of an instruction that caused a jump on a previous pass of the program code), the BTB substitutes the corresponding destination address as the fetch address for the following instruction. When a branch is currently cached and correctly predicted, the performance loss due to branching is reduced from either nine or five stall cycles to zero. For more information, see [“Branch Target Buffer \(BTB\)” on page 8-38](#).

Looping Execution

The sequencer supports zero-overhead and near-zero-overhead looping execution. As shown in [Figure 8-3 on page 8-5](#), a loop lets a program execute a group of instructions repetitively, until a particular condition is met. When the condition is met, execution continues with the next sequential instruction after the loop.

To set up a loop, a program uses a loop counter register, an instruction to decrement the counter, and a conditional instruction jump instruction that tests whether the condition is met and (if not met) jumps to the beginning of the loop.


For zero-overhead loops (no lost cycles for loop test and counter decrement), the sequencer has two dedicated loop counter (LC0 and LC1) registers and special loop counter conditions (counter not expired: IF NLC0E and IF NLC1E, and counter expired: IF LC0E and IF LC1E) for the conditional jump at the end of the loop. Also, the sequencer automatically decrements the counter when executing the special loop counter test and conditional jump. For an example, see [Listing 8-3](#).

Sequencer Operations

Listing 8-3. Zero-Overhead Loop Example

```
LCO = N ;; /* N = 10, sets up loop counter */
_start_loop:
    NOP ;; /* any instruction */
    NOP ;; /* any instruction */
    NOP ;; /* any instruction */
    IF NLCOE, jump _start_loop ;; /* condition test at loop end */
```

Beside the two zero-overhead loops, the sequencer supports any number of near-zero-overhead loops. A near-zero-overhead loop uses an IALU register for the loop counter, includes an instruction to decrement the counter, and uses a condition to test the decrement operation in the conditional jump at the end of the loop. For an example containing zero-overhead and near-zero-overhead loops, see [Listing 8-8 on page 8-82](#).

 For near-zero-overhead loops, programs get better performance through using an IALU register (rather than a compute block register) for the counter. The reason for this performance difference stems from the difference between compute block and IALU instruction execution in the instruction pipeline. For more information, see [“Instruction Pipeline Operations” on page 8-30](#).

Interrupting Execution

Interrupts are events that cause the core to pause its current process, branch, and execute another process. These events can occur at any time and are:

- Internal to the processor
- External to the processor

Interrupts are intended for:

- Synchronizing core and non-core operation
- Error detection
- Debug features
- Control by applications

Each interrupt has a vector register in the Interrupt Vector Table (IVT) and an assigned bit in the interrupt flags and masks registers (ILAT, IMASK and PMASK). The vector register contains the user-definable address of the interrupt routine that services the interrupt. Some important points about the interrupt vector table include:

- 31 interrupts
- Most interrupts are dedicated
- Four general-purpose interrupts associated with $\overline{\text{IRQ3-0}}$ pins



In the IMASK register, a “1” means the interrupt is *unmasked* (DSP recognizes and services interrupt). A “0” means the interrupt is *masked* (processor does not recognize interrupt). The IMASK, ILAT and PMASK registers all have the same bit definitions

The sequencer manages interrupts using the ILAT, IMASK, and PMASK control registers. These registers appear in [Figure 8-12](#), [Figure 8-13](#), [Figure 8-14](#), and [Figure 8-15](#).

Sequencer Operations

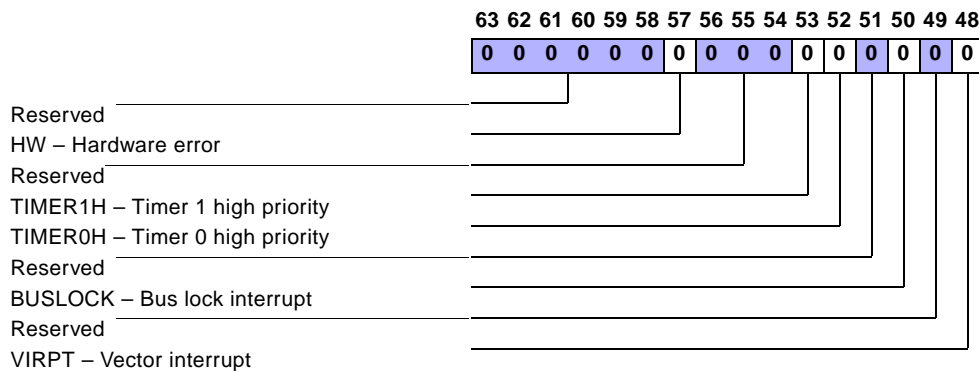


Figure 8-12. IMASKH, ILATH, PMASKH (Upper) Register Bit Descriptions

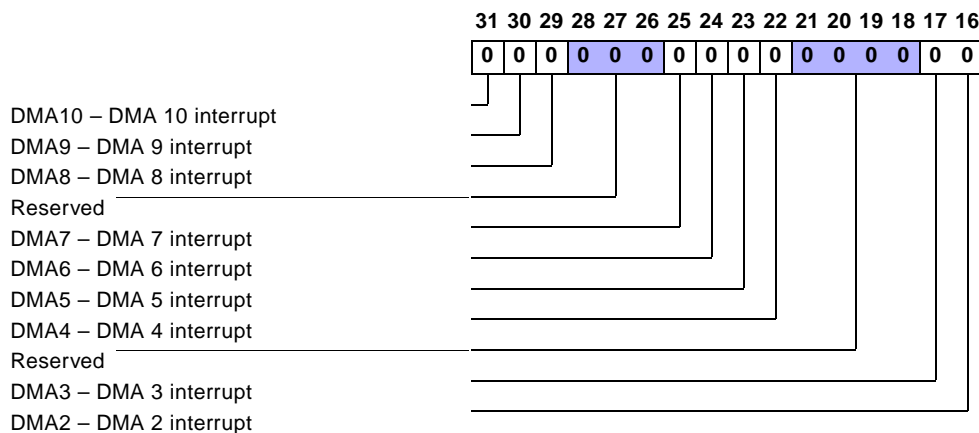


Figure 8-13. IMASKL, ILATL, PMASKL (Upper) Register Bit Descriptions

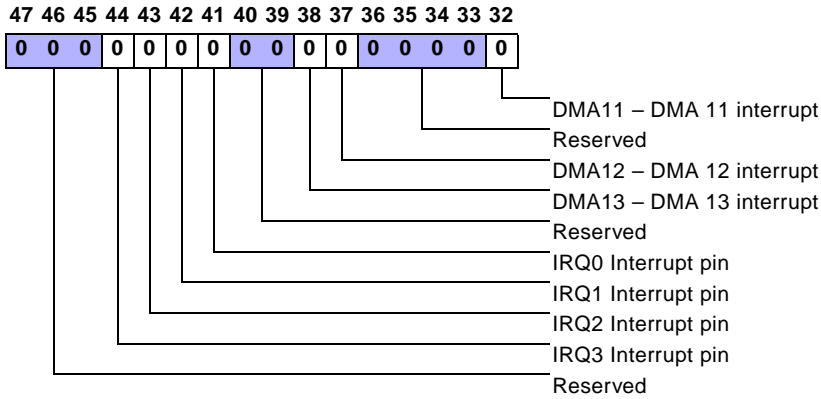


Figure 8-14. IMASKH, ILATH, PMASKH (Lower) Register Bit Descriptions

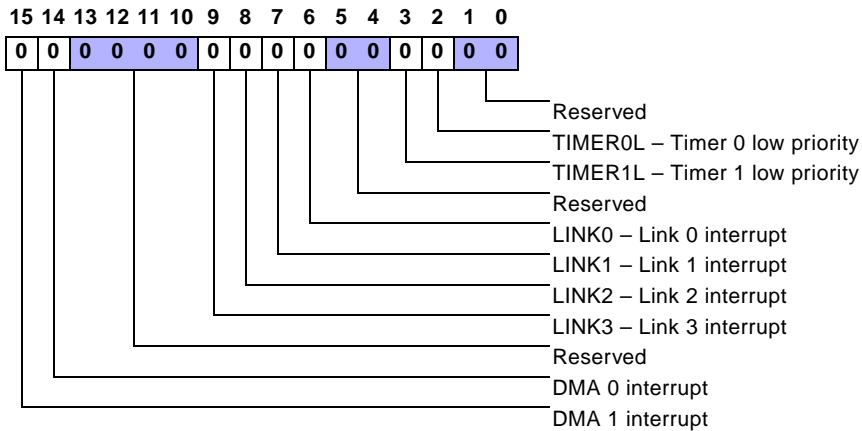


Figure 8-15. IMASKL, ILATL, PMASKL (Lower) Register Bit Descriptions

Sequencer Operations

The ILAT, IMASK, and PMASK control registers have the following usage:

- ILAT – Interrupt Latch Register, latches edge sensitive interrupts (interrupt's bit =1) and displays active level sensitive interrupts (interrupt's =1)
- IMASK – Interrupt Mask Register, masks each individually (interrupt's bit =0) or all interrupts can be masked using the GIE bit in the SQCTL register.
- PMASK – Interrupt Mask Pointer Register, indicates each interrupt being serviced (interrupt's bit =1)

The priority of interrupts high to low matches their order of appearance in IMASK, ILAT, and PMASK. For more information on interrupt sensitivity, interrupt latching, the Interrupt Vector Table, and other interrupt issues, see the *ADSP-TS201 TigerSHARC Processor Hardware Reference*.

Interrupts are classified as either edge or level sensitive. Edge sensitive interrupts are latched when they occur and remain latched until serviced or reset by an instruction. Level sensitive interrupts differ in that if the interrupt is not serviced before the request is removed, the interrupt is forgotten, and if the request remains after the service routine executes, it is considered a new interrupt.

In the sequencer, there are two registers that provide control for interrupt, flag, and timer pins. The flag control (FLAGREG) and interrupt control (INTCTL) registers appear in [Figure 8-16](#), [Figure 8-18](#), [Figure 8-17](#), and [Figure 8-19](#). These registers support:

- **Interrupt sensitivity.** Using the interrupt edge (IRQ_EDGE) bit, programs select edge or level sensitivity for the IRQ3-0 pins independently.
- **Flag control and status.** Using the flag enable (FLAGx_EN), flag output (FLAGx_OUT), and flag input (FLAGx_IN) bits, programs can select whether a flag pin (FLAG3-0) is an input or an output. If an output, select 1 or 0 for the output value. If an input, observe the input value.¹
- **Timer control.** Using the timer run (TMR0RN and TMR1RN) bits, programs can turn on the two timers independently.

The sequencer supports interrupting execution through hardware interrupts (external IRQ3-0 pins and internal process conditions) and software interrupts (program sets an interrupt's latch bit). [Figure 8-3 on page 8-5](#) provides a high-level comparison of branching variations.

¹ The flag pin bit updates the corresponding flag pin after a delay of between 1 and 3 SCLK cycles. Setting and clearing a flag bit might not affect the pin if both operations occur during that delay. The recommended way to set and clear the flag pin bit is to get an external indication that the bit has been set before clearing it. Otherwise, insert (4 x LCLKRAT) instruction lines between the set and the clear to compensate for the delay.

Sequencer Operations

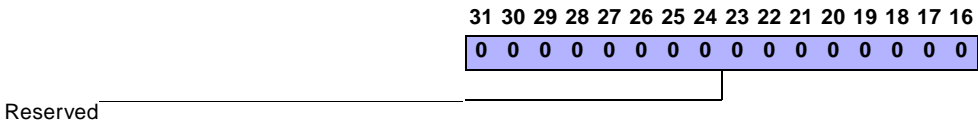


Figure 8-16. FLAGREG (Upper) Register Bit Descriptions

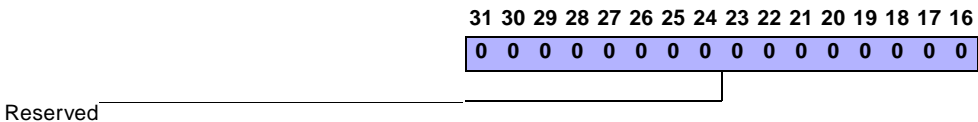


Figure 8-17. INTCTL (Upper) Register Bit Descriptions

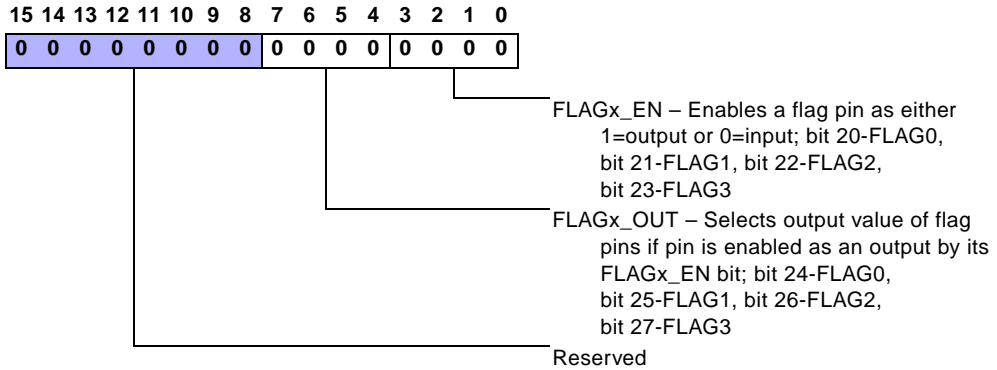


Figure 8-18. FLAGREG (Lower) Register Bit Descriptions

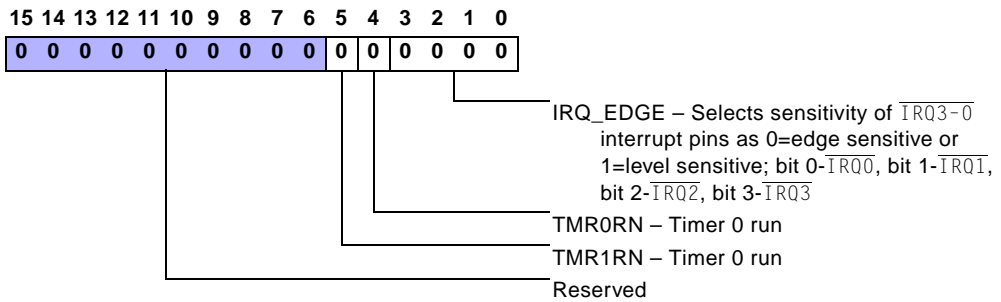


Figure 8-19. INCTL (Lower) Register Bit Descriptions

Sequencer Operations

Looking at [Figure 8-20](#), note some additional details about interrupt service routines for non-reusable interrupts and reusable interrupts. The difference between these two types of interrupt service is that a non-reusable interrupt service routine cannot re-latch the same interrupt again until it has been serviced, and a reusable interrupt service routine (because it has been reduced to subroutine status with the `RDS` instruction) can re-latch the same interrupt while it is still being serviced.

To understand the difference better, note the steps for interrupt processing of these two types of interrupt service.

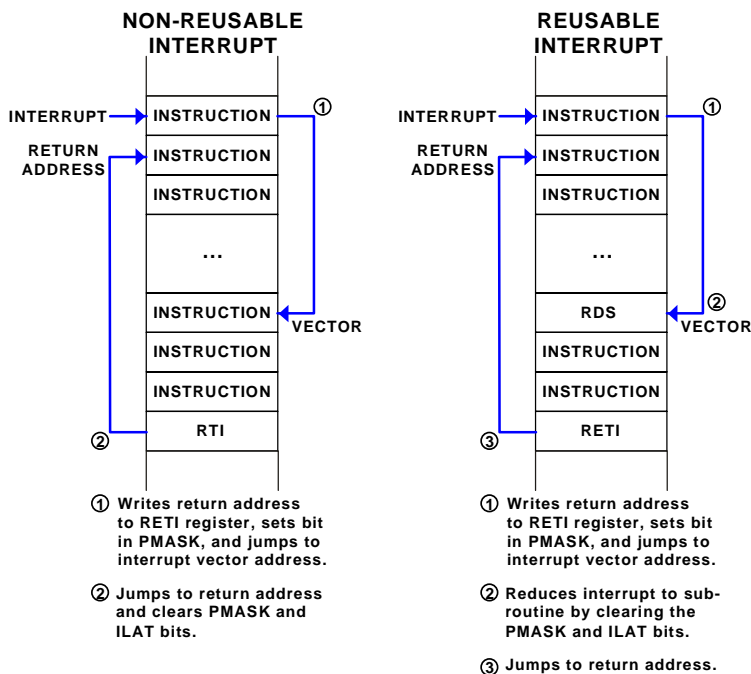


Figure 8-20. Non-Reusable Versus Reusable Interrupt Service

For non-reusable interrupt service, the steps are:

1. Sequencer recognizes interrupt when the interrupt's bit is latched in the `ILAT` register, the interrupt is unmasked by the interrupt's bit in the `IMASK` and `PMASK` registers, and interrupts are globally enabled by the `GIE` bit in the `IMASK` register.
2. Sequencer places the processor in supervisor mode, jumps execution to the interrupt's vector address set in the interrupt's vector register, loads the return address (next sequential address after the interrupt) into the `RETI` register, and sets the interrupt's `PMASK` bits.
3. The processor executes interrupt service routine.
4. Sequencer jumps execution to the return address on reaching the `RTI` instruction at the end of the interrupt service routine, clears the interrupt's `ILAT` bit, and clears the interrupt's `PMASK` bits.


For reusable interrupt service, the steps are:

1. Sequencer recognizes interrupt when the interrupt's bit is latched in the `ILAT` register, the interrupt is unmasked by the interrupt's bit in the `IMASK` and `PMASK` registers, and interrupts are globally enabled by the `GIE` bit in the `IMASK` register.
2. Sequencer places the processor in supervisor mode, jumps execution to the interrupt's vector address set in the interrupt's vector register, loads the return address (next sequential address after the interrupt) into the `RETI` register, and sets the interrupt's `PMASK` bits.
3. Sequencer reduces the interrupt to a subroutine on reaching the `RDS` instruction at the beginning of the interrupt service routine, clears the interrupt's `ILAT` bit, and clears the interrupt's `PMASK` bits.

Because the interrupt's `ILAT` and `PMASK` bits are cleared, the interrupt may be latched again before the interrupt service is completed.

Instruction Pipeline Operations

4. The processor executes interrupt service routine.
5. Sequencer jumps execution to the return address on reaching the RETI instruction at the end of the interrupt service routine.

 Depending on the instruction that is being executed when the interrupt is recognized, different pipeline effects may occur. For more information, see [“Instruction Pipeline Operations” on page 8-30](#).

Instruction Pipeline Operations

As introduced in the instruction pipeline discussion [on page 8-4](#), the processor has a ten-stage instruction pipeline. The pipeline stages and their relationship to the branch target buffer and instruction alignment buffer appears in [Figure 8-4 on page 8-6](#). To better understand the flow of instructions through the pipeline and the time required for each stage, this section uses a series of diagrams similar to [Figure 8-21](#).

Looking at [Figure 8-21](#), note that the instruction pipeline stages appear on the vertical axis and CCLK (processor core clock) cycles appear on the horizontal axis. Usually, each pipeline stage requires one cycle to process an instruction line of up to four instructions. This section describes the situations in which a pipeline stage may require more time to complete its operation. Also note from [Figure 8-21](#) the order in which results from an operation become available. Because IALU arithmetic operations execute at the integer stage and compute block operations execute at the execute 2 stage, IALU results may be available before compute block results, depending on the order and type of instructions.

The first four stages of the instruction pipeline (F1, F2, F3, and F4) are called the *fetch unit pipe*. The fetch cycles are the first pipeline and are tied to the memory accesses. The progress in this pipeline is memory driven and not instruction driven. The fetch unit fills up the instruction alignment buffer (IAB) whenever the IAB has less than three quad words. Since

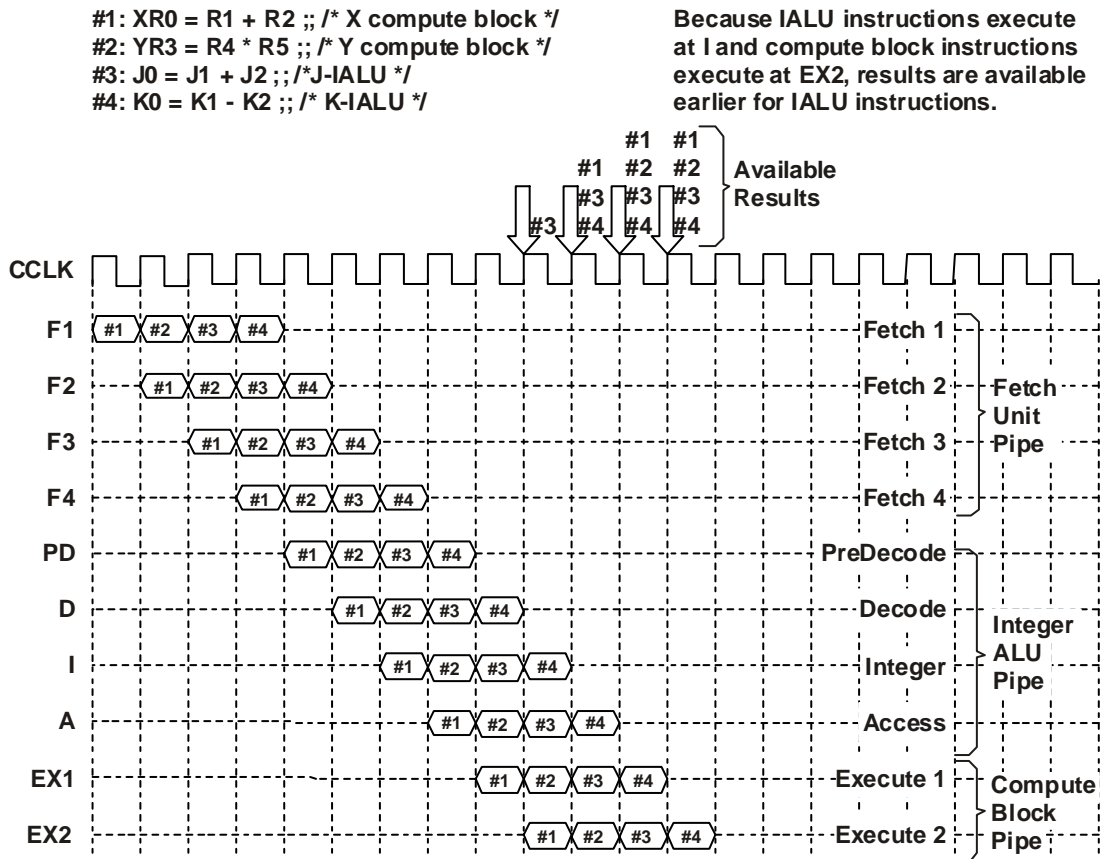


Figure 8-21. Timing for Stages of Instruction Pipeline

the execution units can pull in instructions in throughput lower or equal to the fetch throughput of four words every cycle, it is probable that the fetch unit will fill the IAB faster than the execution units pull the instructions out of the IAB. The IAB can be filled with up to five quad words of instructions.

Instruction Pipeline Operations

When the fetch is from external memory the flow is similar although much slower. The maximum fetch throughput is one instruction line to every two SCLK cycles and the latency is according to the system design (external memory pipeline depth, wait cycles).

The next four instruction pipeline stages (PD, D, I, and A) are called the *integer ALU pipe*. The PreDecode and Decode stages are the first stages in this pipe. In the PreDecode cycle the next full instruction line is extracted from the instruction alignment buffer, and the different instructions are distributed to the execution units during the Decode stage. The units include:

- J-IALU or K-IALU – integer instructions, load/store and register transfers
- Compute block X or Y or both – two instructions (the switching within the compute block is done by the register file)
- Sequencer – branch and condition instructions, and others

The instruction alignment buffer (IAB) also calculates the program counter of a sequential line and some of the non-sequential instructions. The IAB does not perform any decoding. After the Decode stage, instructions enter the integer stage of the integer ALU pipe. IALU instructions include address or data calculation and, optionally, memory access.

Figure 8-21 shows the instruction flow in the IALU pipeline. The IALU instruction's calculation is executed at the Integer stage. If the IALU instruction includes a memory access, the bus is requested on the Integer stage. In this case, the memory access begins at the Access stage as long as the bus is available for the IALU.

The result of the address calculation is ready at the Integer stage. Since the execution of the IALU instruction may be aborted (either because of a condition or branch prediction), the operand is returned to the destination register only at the end of EX2. The result is passed through the pipeline, where it may be extracted by a new instruction should it be

required as a source operand. Dependency between IALU calculations normally do not cause any delay, but there are some exceptions. The data that is loaded, however, is only ready in the register at pipe stage EX2.

The final pipeline stages (EX1 and EX2) are called the *compute block pipe*. The compute block pipe is relatively simple. At the decode cycle, the compute block gets the instruction and transfers it to the execution unit (ALU, multiplier or shifter).

At the Integer stage, the instruction is decoded in the execution unit (ALU, multiplier or shifter), and dependency is checked. At the Access stage, the source registers are selected in the register file. At the execution stages EX1 and EX2, the result and flag updates are calculated by the appropriate compute block. The execution is always two cycles, and the result is written into the target register on the rising edge after pipe stage EX2.

All results are written into the target registers and status flags at pipe stage EX2. There are two exceptions to this rule:

- External memory access, in which the delay is determined by the system
- Multiply-accumulate instructions, which write into MR registers and sticky flags one cycle after EX2 (This write is important to retain coherency in case of a pipeline break.)


When executed either at the Integer stage (IALU arithmetic) or execute 2 stage (all other instructions), the instructions in a single line are executed in parallel. When there are two instructions in the same line which use the same register (one as operand and the other as result), the operand is determined as the value of the register *prior* to the execution of this line. For example:

```
/* Initial values are: R0 = 2, R1 = 3, R2 = 3, R3 = 8 */
R2 = R0 + R1 ; R6 = R2 * R3 (I) ;;
```

Instruction Pipeline Operations

In the previous example, $R2$ is modified by the first instruction, and the result is 5. Still the second instruction sees input to $R2$ as 3, and the result written to $R6$ is 24. This rule is not guaranteed for memory store instructions. In this next example with the same initial values, the result of the first slot is used in the second slot with unpredictable results.

```
R2 = R0 + R1 ; [Address] = R2 ;;  
/* The results of using R2 as input for the memory store instruction here are unpredictable due to possible memory access stalls. The assembler flags this instruction as illegal. */
```

 For best results, do not use the results from one instruction as an operand for another instruction within the same instruction line.

The pipeline creates complications because of the overlap between the execution time of instructions of different lines. For example, take a sequence of two instruction lines where the second instruction line uses the result of the first instruction line as an input operand. Because of the pipeline length, the result may not be ready when the second instruction fetches its operands. In such a case, a *stall* is issued between the first and second instruction line. Since this may cause performance loss, the programmer or compiler should strive to create as few of these cases as possible. These combinations are legal however, and the result will be correct.

This type of problem is discussed in detail in [“Dependency and Resource Effects on Pipeline” on page 8-59](#).

Instruction Alignment Buffer (IAB)

The IAB is a five quad-word FIFO as shown in [Figure 8-22](#). When the sequencer fetches an instruction quad word from memory, the quad word is written into the next entry in the IAB. If there is at least one full instruction line in the IAB, the sequencer can pull it for execution.

The IAB provides these services:

- Buffer the input (fetched instructions) from the fetch unit pipe, keeping the fetch unit independent from the rest of the instruction pipeline. This independence lets the fetch unit continue to run even when other parts of the pipeline stall.
- Align the input (unaligned quad words) to prepare complete instruction lines for execution. This alignment guarantees that complete instruction lines with one to four instructions are able to execute in parallel.

Instructions are 32-bit words that are stored in memory without regard to quad-word alignment (128 bits) or instruction line length (one to four instructions). There is no wasted memory space. Instructions are executed in parallel as determined by the MSB of the opcode for each instruction.

- Distribute the instruction lines to the execution units—IALUs, compute blocks, and sequencer.

Through these services, the IAB insures execution of an entire instruction line without inserting additional stall cycles or forcing memory quad-word alignment on instruction lines.

Instruction Pipeline Operations

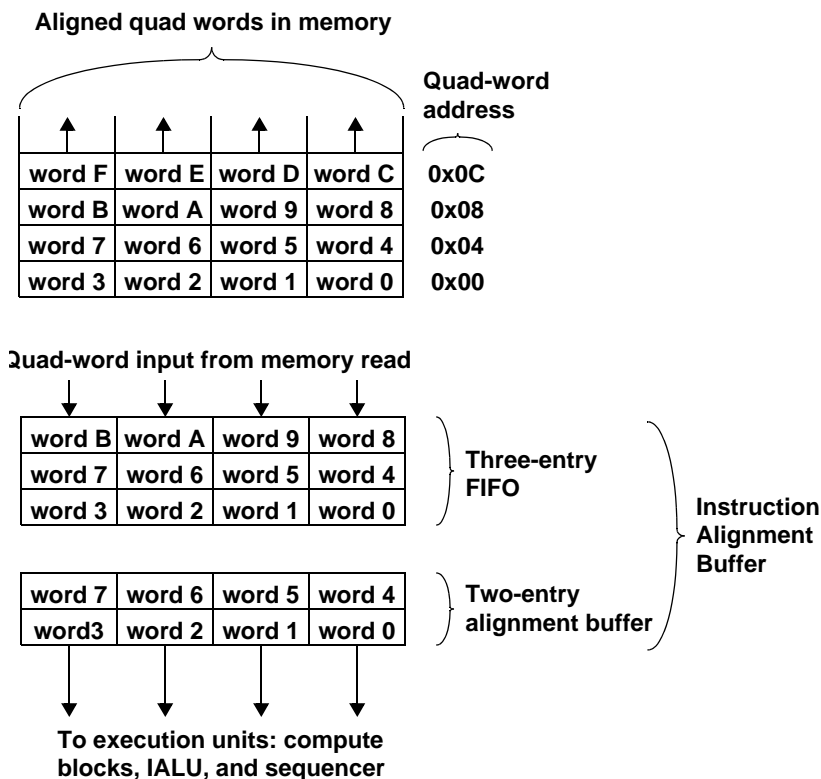


Figure 8-22. Instruction Alignment Buffer (IAB) Structure

To understand the value of the buffer, align, and distribute service that the IAB provides, see [Figure 8-23](#). This figure provides an example of how normal (32-bit) instruction words are stored unaligned in memory and how the IAB buffers, aligns, and distributes these words.

i These descriptions of IAB operation apply to internal memory fetches only. Instruction fetches from external memory result in a much slower instruction flow. Fetch throughput is one instruction

for every SCLK cycle—at best, 25% of the rate for internal memory fetches. Latency depends on system design (external memory pipeline depth, wait cycles width, and other issues).

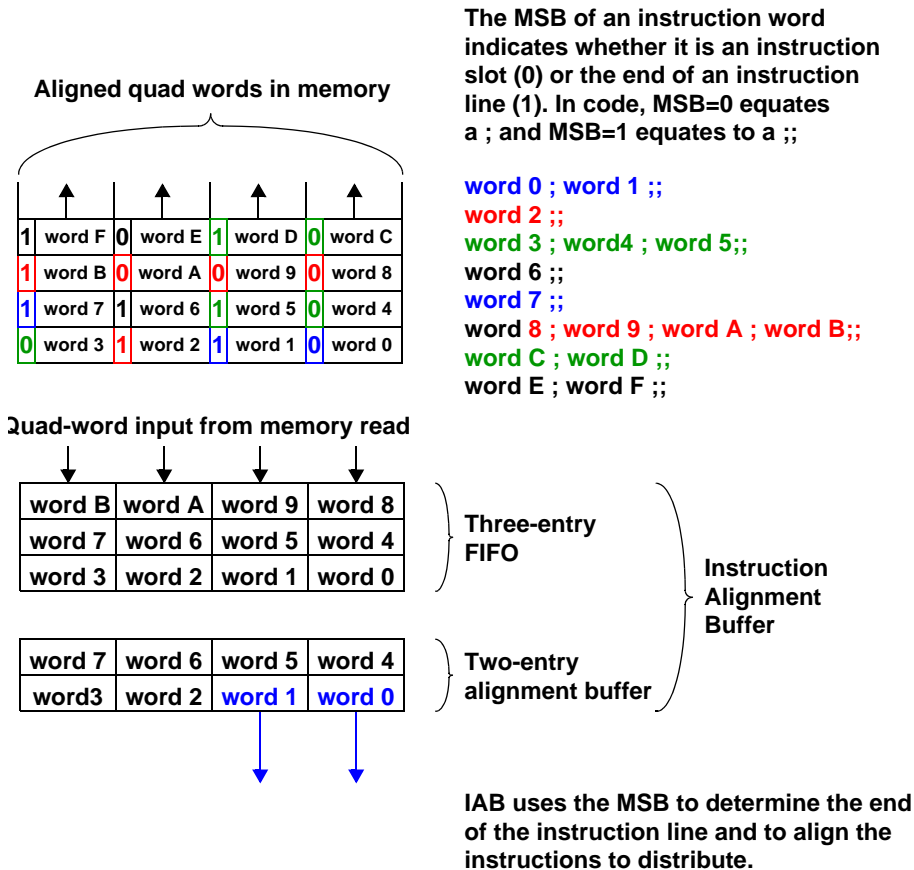


Figure 8-23. IAB Aligns Instruction Lines for Distribution/Execution


Branch Target Buffer (BTB)

The sequencer uses the branch target buffer (BTB) to reduce or eliminate the branch costs (lost cycles as pipeline accommodates the branch) that result from branching execution in the instruction pipeline. The BTB has 32 entries of 4-way set-associative cache (total of 128 entries) that store branch target addresses and has a Least Recently Used (LRU) replacement policy.

The BTB is active after being enabled with the `BTBEN` instruction. For more information, see the [“BTB Enable/Disable” on page 10-236](#).

For permanent buffered program sections, program must lock the BTB using the `BTBLOCK` instruction. While locked, the BTB puts every new entry into the BTB in locked status. When this happens the BTB entry is not replaced until the whole BTB is flushed, in order to keep performance-critical jumps in BTB.

Whenever program overlays are used to swap program segments into and out of internal memory, the BTB must be cleared using the `BTBINV` instruction in order to invalidate the BTB.

 The BTB contents can be accessed directly for debug and diagnostic purposes only, but it must be disabled prior to access by using the `BTBDIS` instruction. Do not directly access BTB contents during normal operations because this access may result in multi-hit and coherency problems.

The BTB structure appears in [Figure 8-24](#). This structure consists of three types of registers that make up each of the 128 BTB entries. The entries are divided into 32 sets and within each set there are four ways. The parts of the BTB include set, way, tag, target, and LRU.

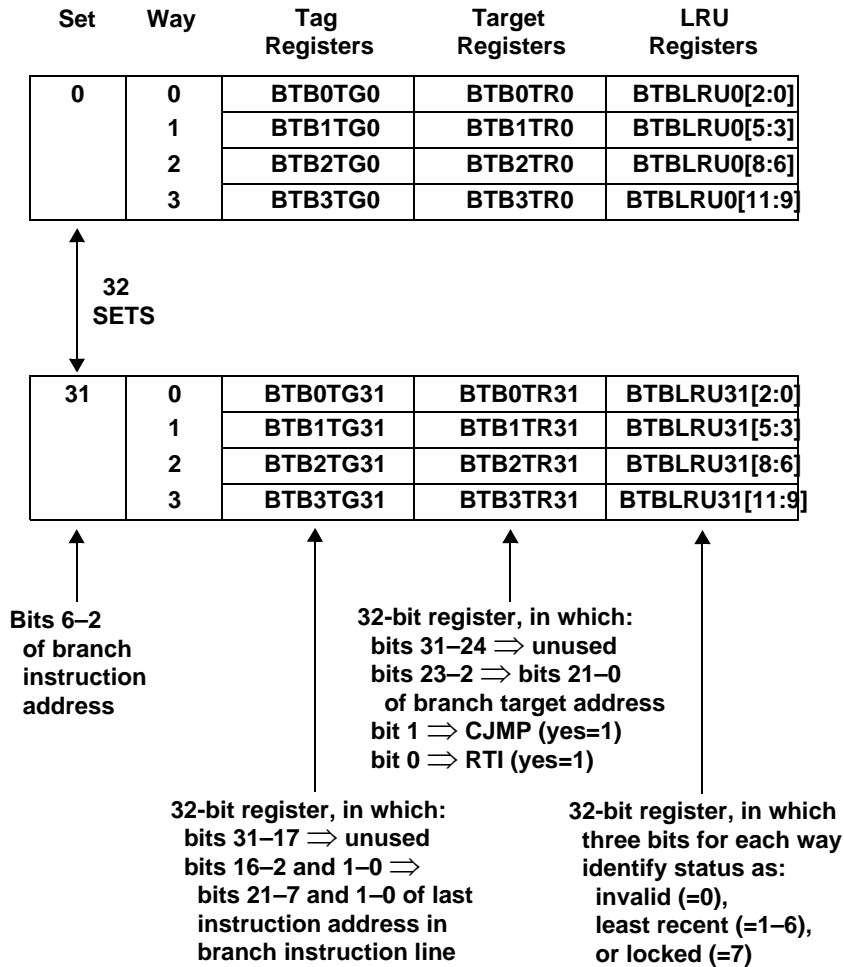


Figure 8-24. Branch Target Buffer (BTB) Structure

Index The index determines the set for the BTB entry. The index is bits 6–2 of the branch instruction's address. Using five bits provides 32 *sets*. Each set contains four *ways* (branch instruction addresses with the same 6–2 bits).

Instruction Pipeline Operations

Tag	The tag determines whether a fetched instruction is a BTB hit (address is in BTB) or a BTB miss (address is not in BTB) when comparing the fetched instruction with contents of ways in the appropriate set. Tags are recorded in the BTB_{ITGxx} registers where i represents the way and xx represents the set. Each register is 32-bits wide, but only the 17 LSBs are valid. These 17 bits are comprised of bits 21–7 and 1–0 of the address of the last instruction in the branch instruction's instruction line.
Target	The target is the target address of the branch. The PC begins fetching at the target address on a BTB hit. Targets are recorded in the BTB_{ITRxx} registers where i represents the way and xx represents the set. Each register is 32-bits wide, but only the 24 LSBs are valid. These 24 bits are comprised of bits 21–0 of the address of the instruction that is the target of the branch, a $CJMP$ bit (0 = branch is not $CJMP$, 1 = branch is $CJMP$), and an RTI bit (0 = branch is not RTI , 1 = branch is RTI),
LRU	The LRU is the Least Recently Used field, which determines whether a way is locked or valid and whether a way should be overwritten in the event of a BTB miss. LRU bits are recorded in the BTB_{LRUxx} registers where xx represents the set. Each register is 32-bits wide, but only the 12 LSBs are valid. Each set has an LRU register that contains a three-bit field for the LRU value for each of the ways in the set. The LRU values represent: 0 = invalid, 1 through 6 = regular LRU value with 6 being most recently used, and 7 = locked.

Now that the controls for the BTB (bits in the `SQCTL` register) and the structure of the BTB (set, way, tag, target, LRU) are understood, it is important to look at how the BTB and branch prediction relate to the instruction pipeline. A flow chart of the sequencer's BTB and branch prediction operations with related branch costs (penalty cycles) appears in [Figure 8-25](#).

As shown in [Figure 8-25](#), BTB usage and branch prediction are independent. The sequencer always compares the fetched instruction against the BTB contents when the instruction is at pipeline Fetch 1 stage. If there is a BTB miss, the branch prediction tells the sequencer at later stages what to assume and when to make its decision. At pipeline Decode stage, the sequencer knows that the instruction is a branch and whether it is a predicted branch or a not predicted (NP) branch. This stage is where the pipeline makes the decision on where to fetch from next. This prediction also affects whether the branch is entered into the BTB, so that a hit occur for the next iteration.



By default, all conditional branches are predicted branches (predicted as taken) and are unconditional branches (treated as though prefixed with the condition `IF TRUE`). Only conditional branches with the (NP) (not predicted) option are predicted as not taken.

The penalty cycles (stalls) for incorrectly predicted branches include:

- When a branch is predicted and there is a BTB hit, the sequencer assumes that the branch is taken and begins fetching from the branch target address when the branch goes from pipeline stage Fetch 1 to Fetch 2. If predicted correctly, this case has zero penalty cycles.
- When a branch is predicted but there is a BTB miss (BTB disabled or no entry match), the sequencer assumes that the branch is taken and begins fetching from the branch target when the branch goes from pipeline stage Predecode to Decode. If predicted correctly, this case has four penalty cycles.

Instruction Pipeline Operations

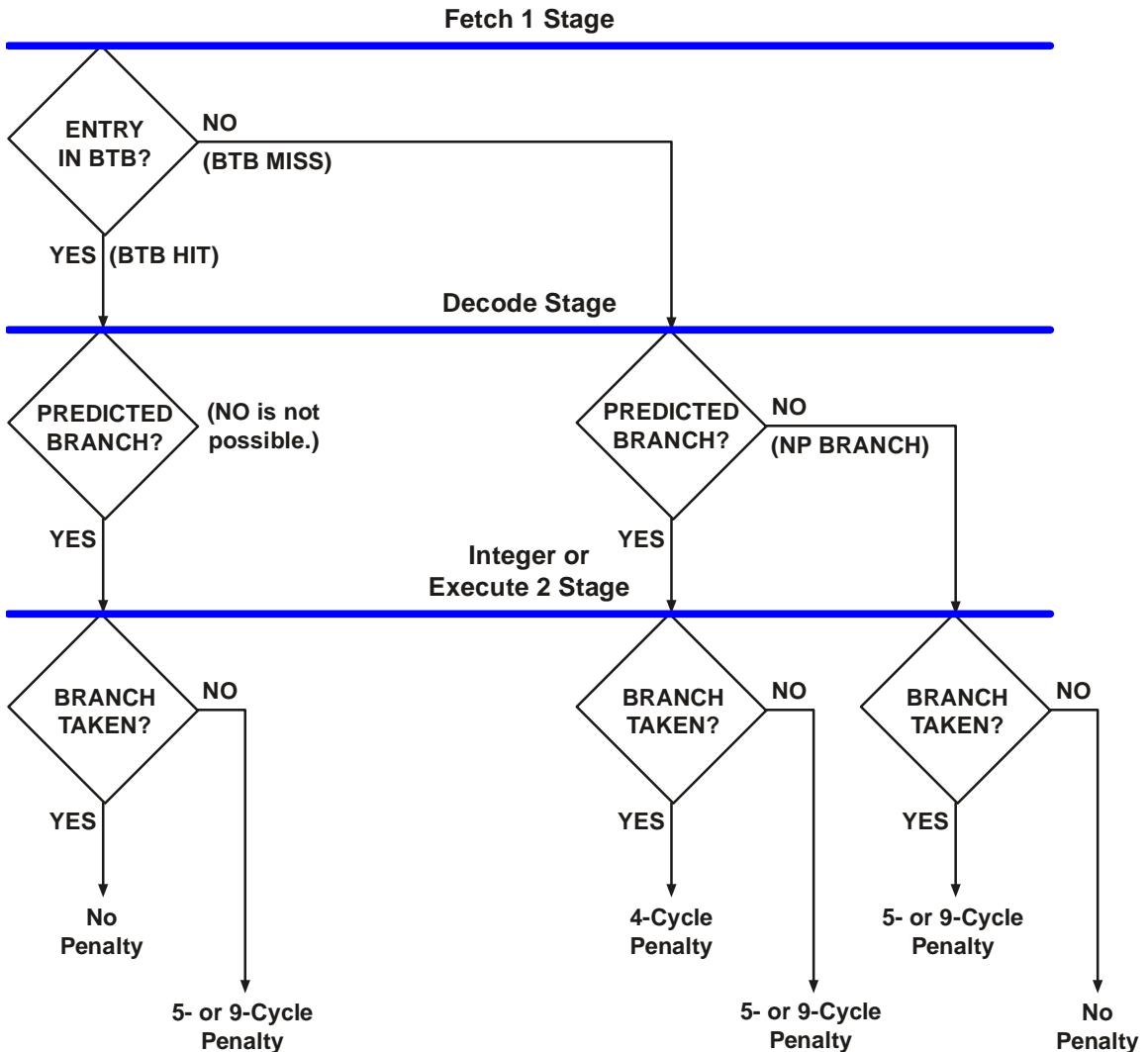


Figure 8-25. Branch Prediction Penalty Tree¹

¹ When the fetched instruction line crosses a quad-word boundary, add one penalty cycle in all cases.

- When a conditional branch is not predicted (NP) and there is a BTB miss (this is always the case with (NP) branches), the sequencer assumes that the branch is not taken and does not begin fetching from the branch target address until the condition is evaluated. This evaluation occurs when the branch goes from pipeline stage Decode to Integer for IALU conditions or from pipeline stage Execute 1 to Execute 2 for compute conditions. If predicted correctly, this case has zero penalty cycles.
- When a conditional branch is not correctly predicted (a predicted branch is not taken or a not predicted (NP) is taken), the sequencer does not catch the incorrect prediction until the condition is evaluated. This evaluation occurs when the branch goes from pipeline stage Decode to Integer for IALU conditions or from pipeline stage Execute 1 to Execute 2 for compute conditions. This case has five (for IALU condition) or nine (for compute condition) penalty cycles.



When the fetched instruction line crosses a quad-word boundary, add one penalty cycle in all cases. Using the `.align_code 4` assembler directive to quad-word align *labels* for JUMP and CALL instructions eliminates this penalty cycle for these branch instructions.

Even assuming that the BTB were disabled (BTBEN bit =0) which would always cause a BTB miss, branch prediction still has an effect on the instruction pipeline's decision making (see all the BTB miss cases above).

Besides understanding the limit of the BTB's effect on the pipeline, it is important to understand the limits of BTB usage regarding branch instruction placement in memory.

Only internal memory branches are cached in the BTB. The width of the cached target addresses is 22 bits. The BTB stores only one tag entry per aligned quad word of program instructions and, consequently, only one branch may be predicted per aligned quad word. If a programmer requires

Instruction Pipeline Operations

more than one adjacent branch be predicted, one to three `NOP` instructions must be inserted between the branches to insure that both branches do not fall into the same aligned quad word. You can also use the `.ALIGN_CODE` assembler directive.

To avoid the possibility of placing more than one instruction containing a predicted branch within the same quad-word boundary in memory and causing unexpected BTB function, this combination of instructions and placement causes an assembler warning. The assembler warns that it has detected two predicted jumps within instruction lines whose line endings are within four words of each other. Further, the assembler states that depending on section alignment, this combination of predicted branch instructions and the instructions placement in memory may violate the constraint that they cannot end in the same quad word.

It is useful to examine how different placement of words in memory results in different contents in the BTB. For example, the code example in [Listing 8-4](#) contains a predicted branch.

Listing 8-4. Predicted Branches, Aligned Quad Words, and the BTB

```
nop; nop; nop; nop;;  
jump HERE; nop;;  
nop; nop; nop; nop;;
```

In memory, each instruction occupies an address, and sets of four locations make up a quad word. The placement of quad words in memory is shown in [Figure 8-22 on page 8-36](#) and discussed in “[Instruction Alignment Buffer \(IAB\)](#)” on page 8-34. The quad-word address is the address of the first instruction in the quad word.

Depending on how the code in [Listing 8-4](#) aligns in memory, quad-word address `0x00000004` could contain:

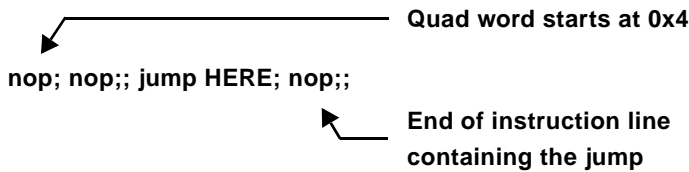


Figure 8-26. Quad Words and Jump Instructions (1)

If so, the BTB entry for the branch would contain:

Tag = 0x00000004, Target Address = HERE

But, the code in [Listing 8-4](#) could align in memory differently. For example, this code could align such that quad-word address 0x00000004 (first line) and 0x00000008 (second line) contain:

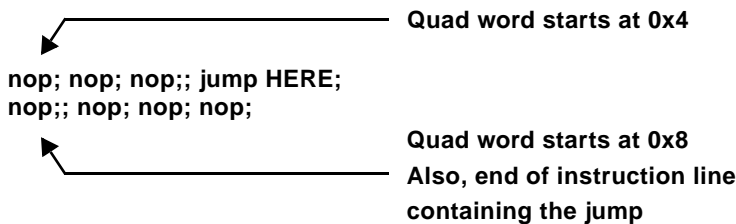


Figure 8-27. Quad Words and Jump Instructions (2)

If so, the BTB entry for the branch would contain:

Tag = 0x00000008, Target Address = HERE

If prediction is enabled, the current PC is compared to the BTB tag values at the F1 stage of the pipeline. If there is a match, the DSP modifies the PC to reflect the branch target address stored in the BTB, and the sequencer continues to fetch subsequent quad words at the modified PC. If there is no match, the DSP does not modify the PC, and the sequencer continues to fetch subsequent quad words at the unmodified PC.

Instruction Pipeline Operations

When the same instruction reaches the Decode stage of the pipeline, the instruction is identified as a branch instruction. If there was a BTB match, no branch-exception action is taken. The PC has already been modified, and the sequencer has already fetched from the branch target address. If there is no BTB match, the sequencer aborts the four instructions fetched prior to reaching the Decode stage (four stall cycles), and the processor modifies the PC to reflect the branch target address and begins fetching quad words at the modified PC. The sequencer updates the BTB with the branch target address such that the next time the branch instruction is encountered, it is likely that there will be a BTB match.

The BTB contents vary with the instruction placement in memory, because:

- The sequencer fetches instructions a full quad word at a time.
- An instruction line may occupy less than a full quad word, occupy a full quad word, or span two quad words.
- An instruction line may start at a location other than a quad-word aligned address.

Because the BTB can store only a single branch target address for each aligned quad word of instruction code, it's important to examine coding techniques that work with this BTB feature. The following code example produces unpredictable results in the hardware, because this code (depending on memory placement) may attempt to force the BTB to store multiple branch target addresses from a single aligned quad word.

```
jump FIRST_JUMP; LC1 = yR16;;  
jump SECOND_JUMP; R29 = R27;;
```

Illegal, the line ends of instruction lines that contain JUMPs are within four instructions of each other.

Figure 8-28. Quad Words and Jump Instructions (3) Illegal Proximity

The situation can be remedied by using NOP instructions to force the branch instructions to exhibit at least four words of separation.

```
jump FIRST_JUMP; LC1 = yR16;;  
jump SECOND_JUMP; R29 = R27; nop; nop;;  
/* Adding NOPs moves the line ending of 2nd instruction */
```

While adding these NOP instructions increases the size of the code, these NOP instructions do not affect the performance of the code.

Another way to control the relationship between alignment of code within quad words and BTB contents is to use the `.align_code 4` assembler directive. This directive forces the immediately subsequent code to be quad-word aligned as follows:

```
jump FIRST_JUMP; LC1 = yR16;;  
.align_code 4;  
/* Forcing quad alignment shifts the line ending of the next  
instruction */  
jump SECOND_JUMP; R29 = R27;;
```

If the BTB hit is a computed jump, the RETI or CJMP register is used (according to the instruction) as a target address. In this case, any change in this register's value until the jump takes place will cause the ADSP-TS201 processor to abort the fetched instructions and repeat the flow as if there were no hit.

Conditional Branch Effects on Pipeline

Correct prediction of conditional branches is critical to pipeline performance. Prediction affects, and is affected by, all the pipes (fetch, IALU, compute) in the pipeline. Each branch flow differs from every other and is derived by the following criteria:

- Jump prediction (See [“Conditional Execution” on page 8-12.](#))
- BTB hit or miss (See [“Branch Target Buffer \(BTB\)” on page 8-38.](#))
- Condition pipe stage—pipeline stage Integer or Execute 2—when is it resolved (See the Branch Prediction Penalty Tree in [Figure 8-25 on page 8-42.](#))

The prediction is set by the programmer or compiler. The prediction is normally `TRUE` or ‘branch taken’. When the programmer uses option (NP) in a control flow instruction, the prediction is ‘branch not taken’. [For more information, see “Branch Target Buffer \(BTB\)” on page 8-38.](#) In general, prediction indicates if the default assumption for this branch will or will not be taken. Take, for example, a loop that is executed n times, where the branch is taken $n-1$ times, and always more than once. Setting this bit has two consequences:

- The branch goes into BTB.
- At stage Decode, the ADSP-TS201 processor identifies the instruction as a jump instruction and continues fetching from the target of the jump, regardless of the condition.

If a branch instruction is a BTB hit, the ADSP-TS201 processor fetches, in sequence, the target of the branch after fetching the branch. In this case there is no overhead for correct prediction. For a detailed description of BTB behavior see [“Branch Target Buffer \(BTB\)” on page 8-38.](#)

The various condition codes are resolved in different stages. IALU conditions are resolved in stage Integer of the instruction that updates the condition flags. Compute block flags are updated in pipe stage EX2. The other flags (BM, FLG0-3, and TRUE) are asynchronous to the pipeline because they are created by external events. These are used in the same fashion as IALU conditions and are resolved at pipe stage Integer, except for the condition BM, which is resolved at pipe stage EX2.

Different situations produce different flows and, as a result, different performance results. The parameters for the branch cost are:

- Prediction – branch is taken or not taken
- Branch on IALU or compute block
- BTB hit – miss
- Branch real result – taken or not

There are 16 combinations. The following combinations are ignored.


- If the prediction is ‘not taken’, the BTB cannot give a hit.
- If the prediction is ‘not taken’ and the branch is not taken, the flow is as if no branch exists.
- If the prediction is ‘taken’ and the branch is taken, the flow is identical for IALU and compute block instructions.

The different flows are shown in [Figure 8-29 on page 8-51](#) through [Figure 8-40 on page 8-74](#). Each diagram shows the flow of each combination and its cost. The cost of a branch can be summarized as:

- Prediction not taken, branch not taken – no cost
- BTB hit, branch taken – no cost

Instruction Pipeline Operations

- BTB miss, prediction taken, branch taken – four cycles
- Prediction taken, branch not taken (either BTB hit or miss); or prediction not taken, branch taken: IALU condition (five cycles) or Compute block (nine cycles)

 If the prediction is ‘not taken’, there cannot be a BTB hit since the ‘prediction taken’ is a condition for adding an entry to BTB.

One cycle should be added to the above branch costs if one of the following applies:

- The jump is taken and the target instruction line crosses a quad-word boundary.
- The branch was predicted to be taken but was not taken, and the sequential instruction line crosses a quad-word boundary.

Figure 8-29 shows a predicted branch that is based on an IALU condition. Because the branch is correctly predicted as taken and the BTB contains the branch instruction (BTB hit), the pipeline contents are continuously executable (no pipeline stages voided), and there are no lost cycles (branch cost). Note that the pipeline evaluates the IALU condition (JEQ flag set by instruction #1) when the conditional instruction reaches the Integer (I) pipeline stage.

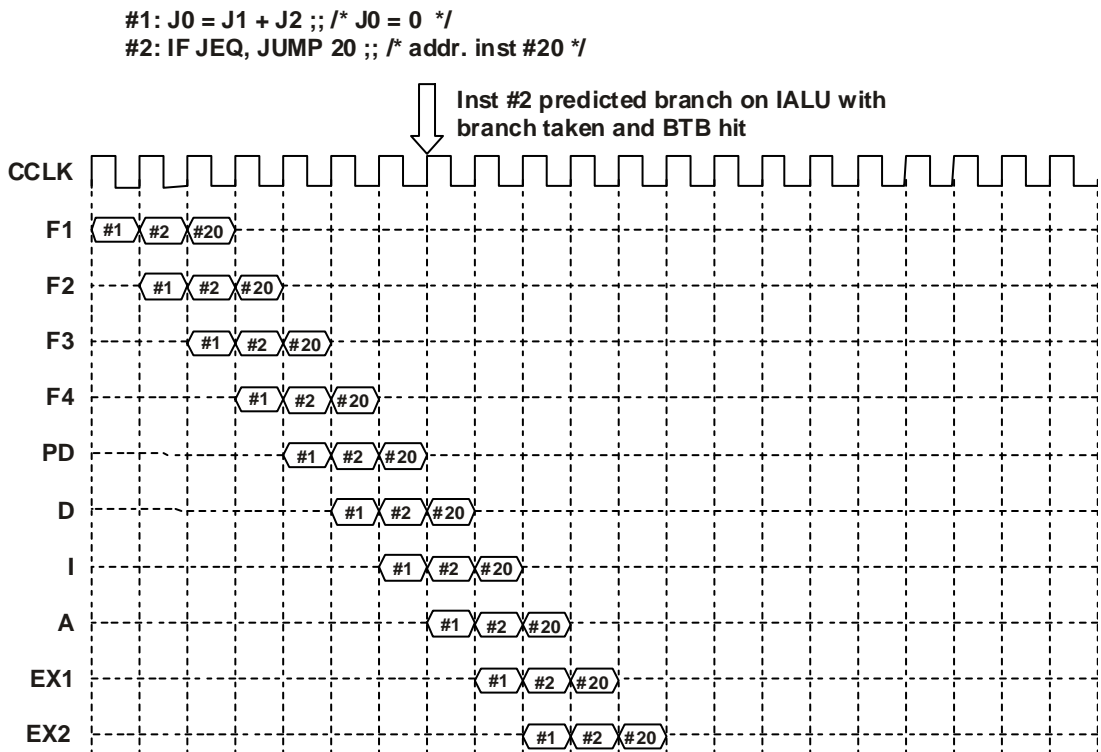


Figure 8-29. Predicted Branch Based on IALU Condition—Branch Taken—With BTB Hit

Instruction Pipeline Operations

Figure 8-30 shows a predicted branch that is based on an IALU condition. Because the branch is correctly predicted as taken and the BTB does not contain the branch instruction (BTB miss), the pipeline contents are not continuously executable (four pipeline stages voided), and there are four lost cycles (branch cost). Note that the pipeline evaluates the IALU condition (JEQ flag set by instruction #1) when the conditional instruction reaches the Integer (I) pipeline stage. From this evaluation, the pipeline determines how many pipeline stages to void.

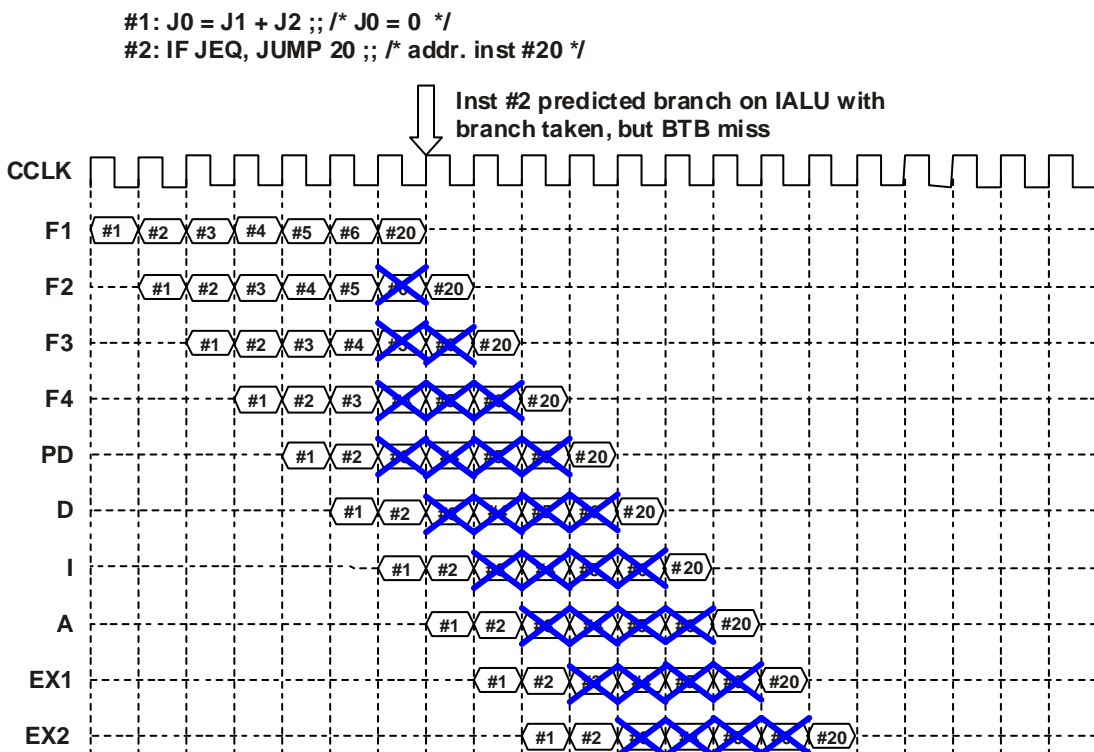


Figure 8-30. Predicted Branch Based on IALU Condition—Branch Taken—With BTB Miss

Figure 8-31 shows a not predicted (NP) branch that is based on a compute condition. Because the branch is incorrectly predicted as not taken, the pipeline contents are not continuously executable (nine pipeline stages voided), and there are nine lost cycles (branch cost). Note that the pipeline evaluates the compute condition (AEQ flag set by instruction #1) when the conditional instruction reaches the Execute 2 (EX2) pipeline stage. From this evaluation, the pipeline determines how many pipeline stages to void.

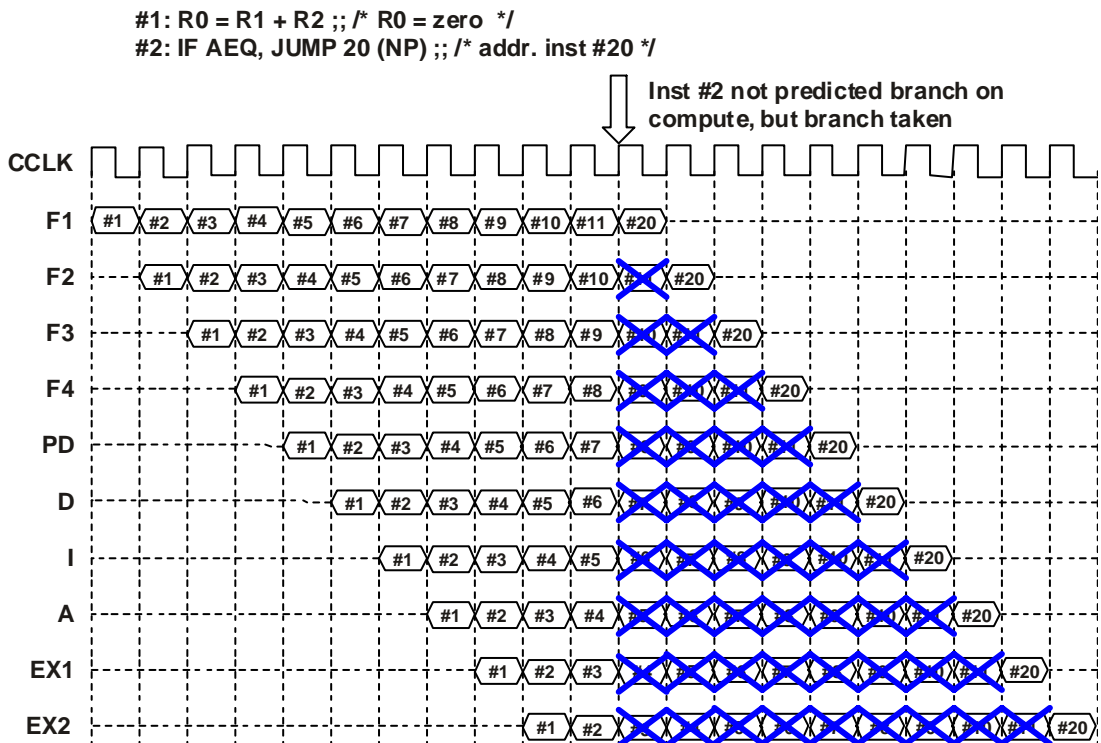


Figure 8-31. Not Predicted (NP) Branch Based on Compute Condition—Branch Taken

Instruction Pipeline Operations

Figure 8-32 shows a not predicted (NP) branch that is based on an IALU condition. Because the branch is incorrectly predicted as not taken, the pipeline contents are not continuously executable (five pipeline stages voided), and there are five lost cycles (branch cost). Note that the pipeline evaluates the IALU condition (JEQ flag set by instruction #1) when the conditional instruction reaches the Integer (I) pipeline stage. From this evaluation, the pipeline determines how many pipeline stages to void.

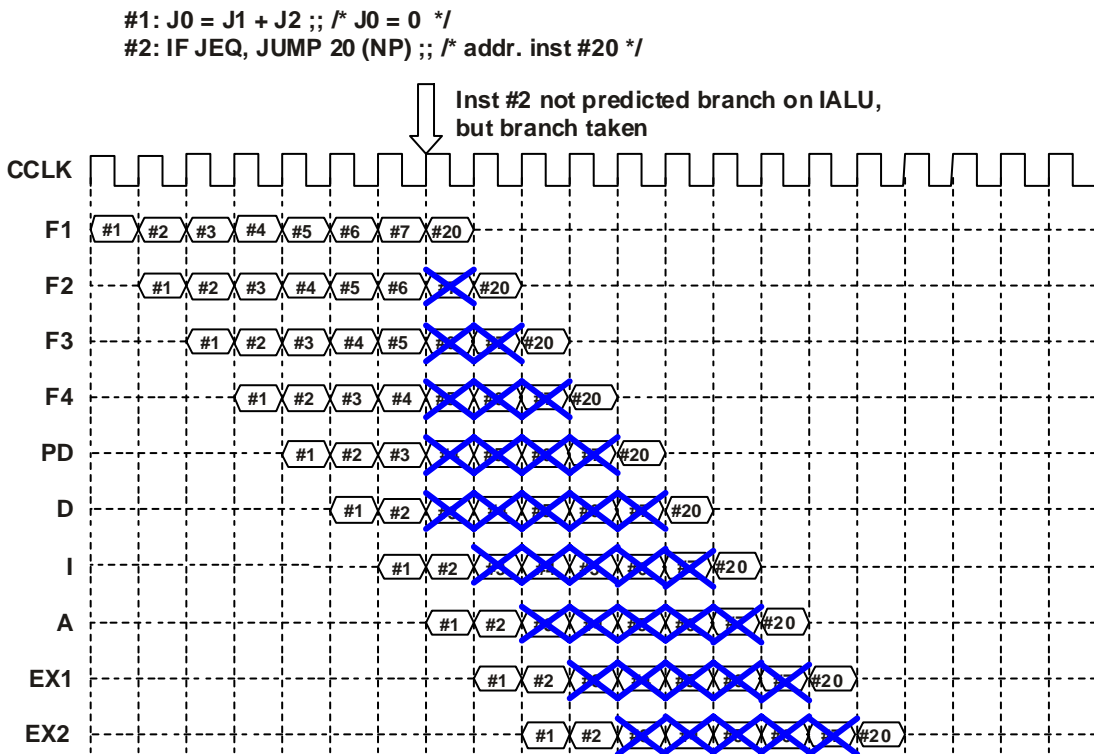


Figure 8-32. Not Predicted (NP) Branch Based on IALU Condition—Branch Taken

Figure 8-33 shows a predicted branch that is based on a compute condition. Because the branch is incorrectly predicted as taken and the BTB contains the branch instruction (BTB hit), the pipeline contents are not continuously executable (nine pipeline stages voided), and there are nine lost cycles (branch cost). Note that the pipeline evaluates the compute condition (AEQ flag set by instruction #1) when the conditional instruction reaches the Execute 2 (EX2) pipeline stage. From this evaluation, the pipeline determines how many pipeline stages to void.

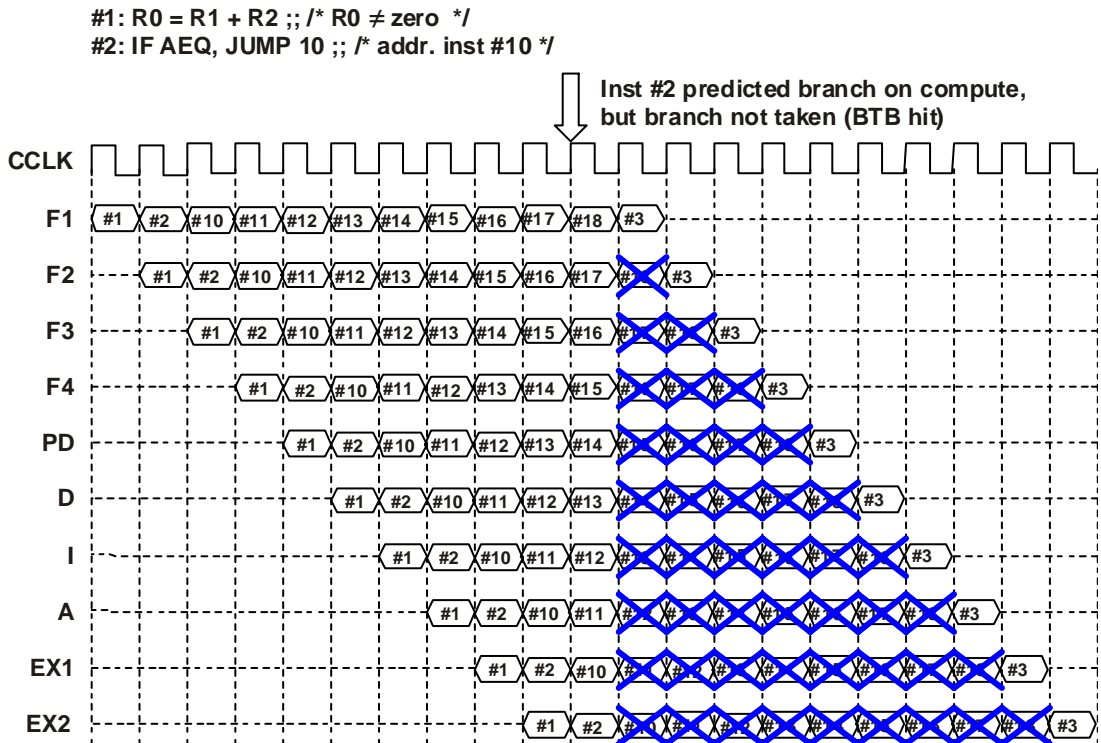


Figure 8-33. Predicted Branch Based on Compute Block Condition—Branch Not Taken—With BTB Hit

Instruction Pipeline Operations

Figure 8-34 shows a predicted branch that is based on an IALU condition. Because the branch is incorrectly predicted as taken and the BTB contains the branch instruction (BTB hit), the pipeline contents are not continuously executable (five pipeline stages voided), and there are five lost cycles (branch cost). Note that the pipeline evaluates the compute condition (JEQ flag set by instruction #1) when the conditional instruction reaches the Integer (I) pipeline stage. From this evaluation, the pipeline determines how many pipeline stages to void.

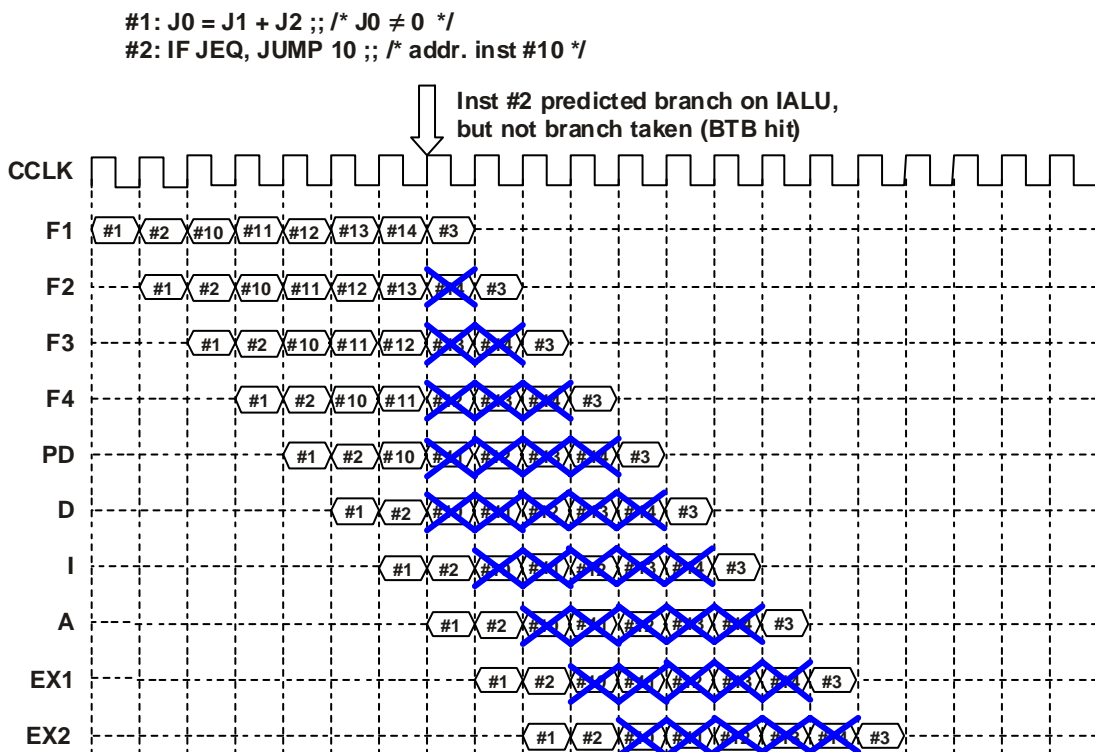


Figure 8-34. Predicted Branch Based on IALU Condition—
 Branch Not Taken—With BTB Hit

Figure 8-35 shows a predicted branch that is based on a compute condition. Because the branch is incorrectly predicted as taken and the BTB does not contain the branch instruction (BTB miss), the pipeline contents are not continuously executable (nine pipeline stages voided), and there are nine lost cycles (branch cost). Note that the pipeline evaluates the compute condition (AEQ flag set by instruction #1) when the conditional instruction reaches the Execute 2 (EX2) pipeline stage. From this evaluation, the pipeline determines how many pipeline stages to void.

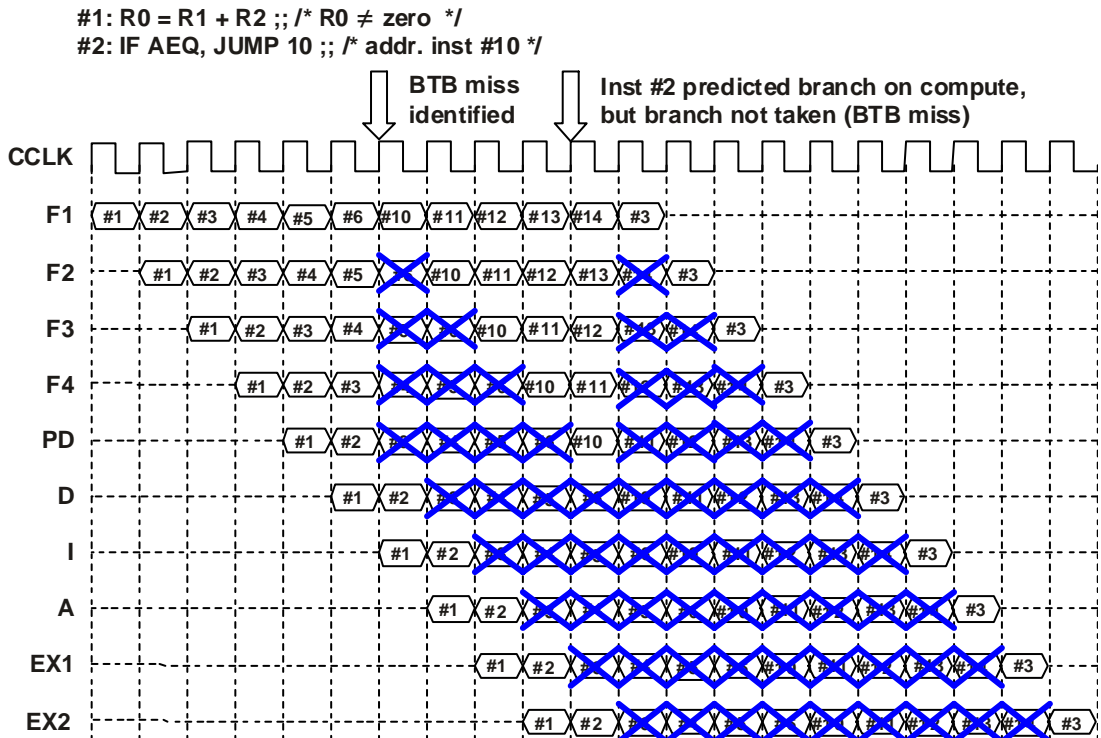


Figure 8-35. Predicted Branch Based on Compute Block Condition—Branch Not Taken—With BTB Miss

Instruction Pipeline Operations

Figure 8-36 shows a predicted branch that is based on an IALU condition. Because the branch is incorrectly predicted as taken and the BTB does not contain the branch instruction (BTB miss), the pipeline contents are not continuously executable (five pipeline stages voided), and there are five lost cycles (branch cost). Note that the pipeline evaluates the compute condition (JEQ flag set by instruction #1) when the conditional instruction reaches the Integer (I) pipeline stage. From this evaluation, the pipeline determines how many pipeline stages to void.

```
#1: J0 = J1 + J2 ;; /* J0 ≠ 0 */
#2: IF JEQ, JUMP 10 ;; /* addr. inst #10 */
```

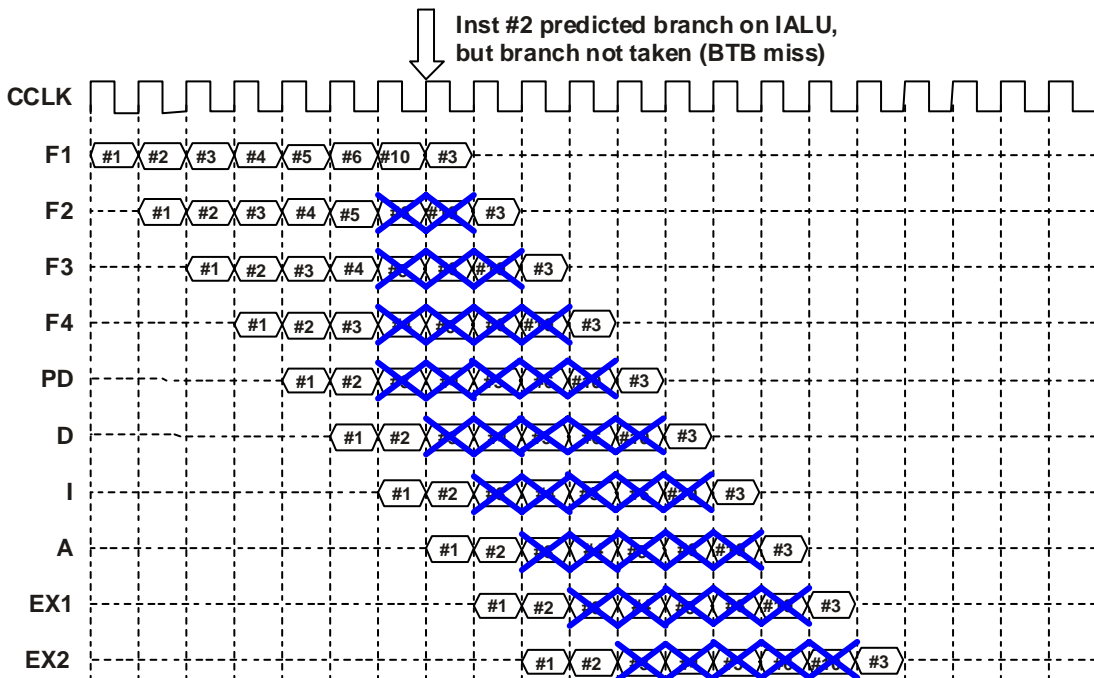


Figure 8-36. Predicted Branch Based on IALU Condition—Branch Not Taken—With BTB miss

Dependency and Resource Effects on Pipeline

The ADSP-TS201 processor supports any sequence of instruction lines, as long as each separate line is legal. The pipelined instruction execution causes overlap between the execution of different lines. Two problems may arise from this overlap.

- Dependency
- Resource conflict

A *dependency condition* is caused by any instruction that uses as an input the result of a previous instruction, if the previous instruction data is not ready when the current instruction needs the operand.

A *resource conflict* is caused by internal bus contention or page cache miss during an internal memory access. The following instructions may cause a resource conflicts.

- Load or store instructions request an internal bus to access an internal memory block. If the address is external, load and store instructions use the virtual bus.
- Immediate load, move register-to-register, and add or subtract with C JMP option instructions all request the virtual bus.


If any of the above two instructions use the same internal bus, or if another resource (DMA or bus interface) requests the same bus on the same cycle that the IALU requests the bus, the bus might not be granted to the IALU. This in turn could cause a delay in the execution.

As shown in [Figure 8-4 on page 8-6](#), memory accesses use a memory pipeline that operates in parallel with the instruction pipeline. During instruction fetches, bus contention conflicts and cache misses appear as execution delays during the address and register cycles of the I-bus memory pipe. During other memory accesses, bus contention conflicts and cache misses appear as execution delays during the address and register

Instruction Pipeline Operations

cycles of the J/K-bus memory pipe. For more information on the memory pipeline and cache, see the *ADSP-TS201 TigerSHARC Processor Hardware Reference*.

This section details the different cases of stalls. A *stall* is any delay that is caused by one of the two conditions previously described. Although the information in this manual is detailed, there may be some cases that are not defined here or conditions that are not always perceivable to the system designer. Exact behavior can only be reproduced through the TigerSHARC processor simulator.

 Some architectural differences lead to execution stalls on the ADSP-TS201 processors that did not occur on previous TigerSHARC processors. The ADSP-TS201 processor uses a 10-stage pipeline, which is two stages deeper than previous TigerSHARC processors. Also, some critical path cases on the ADSP-TS201 processors are resolved with a stall.

In previous TigerSHARC processors, most of the dependency logic operates in pipeline stage Integer. In the ADSP-TS201 processors, most of the dependency logic operates in pipeline stage Decode (for computes) and PreDecode (for IALU). The stall logic on bus transactions operates in pipeline stage Access.

Table 8-1 lists the dependency related stalls for the ADSP-TS201 processors. Unless noted, the data type or size does not affect dependency related stalls

Table 8-1. Instruction Dependencies - Stall List

First Instruction	Second Instruction	Dependency	Stalls	Stage
Rs1=X/Ystat	Any compute instruction Rs=F (Rm2, Rn2);	Rs1=Rm2 or Rs1=Rn2	2	D
Rs1=X/Ystat	[mem address]=Rm2; Ureg=Rm2;	Rs1=Rm2	1	D

Table 8-1. Instruction Dependencies - Stall List (Cont'd)

First Instruction	Second Instruction	Dependency	Stalls	Stage
TRsq1=ACS ();	TRsq2=ACS (TRmd2, TRnd2, Rm);	TRmd2=TRsq1	0	D
STRsq1=ACS ();	STRsq2=ACS (TRmd2, TRnd2, Rm);	TRmd2=TRsq1	0	D
If <cond>; do, <S>TRsq1= ACS/MAX/TMAX();	Any CLU instruction	TRmd2=TRsq1 TRnd2=TRsq1	2	D
If <cond>; do, TRsq1=Rm	TRs+=XCORRS (...); or TRs+=DESPREAD (...);	Always	1	D
STRsq1=ACS ();	TRsq2=ACS (TRmd2, TRnd2, Rm);	TRmd2=TRsq1	1	D
TRsq1=ACS ();	STRsq2=ACS (TRmd2, TRnd2, Rm);	TRmd2=TRsq1	1	D
(S)TRsq1=ACS ();	(S)TRsq2=ACS (TRmd2, TRnd2, Rm);	TRnd2=TRsq1	1	D
TRsq1=ACS();	MAX or TMAX TRsq2=F (TRmd2, TRnd2, Rm);	TRmd2=TRsq1 or TRnd2=TRsq1	1	D
TRsd1=MAX/TMAX();	ACS, MAX or TMAX TRsq2= F(TRmd2, TRnd2, Rm);	TRmd2=TRsq1 or TRnd2=TRsq1	1	D
ACS, TMAX or MAX TRs1=F(.)	Despread or XCORRS TRs2=F(TRm2, TRn2)	Always ¹	1	D
Despread or XCORRS	ACS, TMAX or MAX	Always ¹	2	D
TRs=Rm; or THR _s =Rm; or CMCTL=Rm;	Any CLU instruction	Always	0	D
If <cond>; do, TRs1=Rm; or CMCTL=Rm;	Any CLU instruction Rs/TRs=F(TRm2, TRn2)	TRmd2=TRs1 or TRnd2=TRs1	2	D
If <cond>; do THR _s 1=Rm;	TRs= DESPREAD/XCORRS	Always ¹ (because of THR _r dependency)	2	D

Instruction Pipeline Operations

Table 8-1. Instruction Dependencies - Stall List (Cont'd)

First Instruction	Second Instruction	Dependency	Stalls	Stage
If <cond>; do, TRs= DESPREAD/ XCORRS/ACS ()	TRs= DESPREAD/XCORRS	Always ¹ (because of THr dependency)	2	D
If <cond>; do, CMCTL=Rm;	TR15:0/31:16= XCORRS () (cut R)	Always ¹ (because of CMCTL register dependency)	2	D
Any CLU instruction	Rs=TRm; or Rs=THRm;	Always	0	D
PR+=sum (...);;	if <compute ALU cond> ;;	Always	1	D
Rs=sum Rm;;	if <compute ALU cond> ;;	Always	1	D
PR+=abs ();;	if <compute ALU cond> ;;	Always	1	D
If <cond>; do, Mra+= ; If <cond>; do, Mra -= ; If <cond>; do, Mra=Rm;	Mrb+= ; Mrb -= ; Rm=MRb;	Mra=MRb or MRa ==MR4 or MRb ==MR4	1	D
Any compute instruction Rs1=F (Rm, Rn);	Any compute instruction Rs=F (Rm2, Rn2);	Rs1=Rm2 or Rs1=Rn2	1	D
Any load from internal Rs1=[]; or Rs1=Ureg	Any compute instruction Rs=F (Rm2, Rn2);	Rs1=Rm2 or Rs1=Rn2	1+miss ²	D,A
Any load from external Rs1=[]; or Rs1=SOC_Ureg ³	Any compute instruction Rs2=F (Rm2, Rn2);	Rs1=Rm2 or Rs1=Rn2 or Rs1=Rs2	3+SOC delay ⁴	D,A
Rsq1=dab q [external memory];;	Rsq2=dab q[]	Always	3+SOC delay ⁴	
Any compute instruction Rs1=F (Rm, Rn);	Any store instruction []=Rm; or Ureg=Rm	Always	0	D
Any memory system com- mand register load CACMDx=<Ureg/imm>	Any access to the segment i	Same segment in both instructions	4	I

Table 8-1. Instruction Dependencies - Stall List (Cont'd)

First Instruction	Second Instruction	Dependency	Stalls	Stage
Any memory system register load CACMDx=<Ureg/imm>; CCAIRx=<Ureg/imm>; CACMADRx=<Ureg>; CADATAx=<Ureg/imm>;	Any memory system register store <Ureg>=CACMDx; <Ureg>=CCAIRx; <Ureg>=CACMADRx; <Ureg>=CADATAx; <Ureg>=CASTATx	Same segment in both instructions	4	I
Any internal memory access that causes a miss in cache	Any instruction	Always	Miss penalty	A
Any IALU instruction ⁵ Js1=F (Jm, Jn)	Any IALU instruction ² Js=F (Jm2, Jn2)	Always	0	PD
Any IALU instruction ⁵ if <conde2>; do, Js1=F (Jm, Jn) if (conde2); do, Ureg=[Js1+=] if (conde2); do, [Js1+=]=Ureg	Any IALU instruction ² Js=F (Jm2, Jn2);	Jm2=Js1 or Jn2=Js1	4	PD
Any IALU instruction ⁵ if <conde2>; do, Js1=F (Jm, Jn)	Js=Jm+Jn+Jc; or Js=Jm - Jn+Jc - 1;	Jm2=Js1 or Jn2=Js1	4	PD
Any IALU instruction ⁵ if <conde2>; do, Js1=F (Jm, Jn)	Usage of IALU condition if <JALU condition>	Always	4	PD
Any IALU instruction ⁵ if <condi>; do, Js1=F (Jm, Jn) if (condi); do, Ureg=[Js1+=] if (condi); do, [Js1+=]=Ureg	Any IALU instruction ² Js=F (Jm2, Jn2);	Jm2=Js1 or Jn2=Js1	1	PD
JB=Jm +/- Jn; or JL=Jm +/- Jn; ⁵	Js=Jm2 +/- Jn2 (CB);	JB / JL belongs to JM2	5	PD

Instruction Pipeline Operations

Table 8-1. Instruction Dependencies - Stall List (Cont'd)

First Instruction	Second Instruction	Dependency	Stalls	Stage
JB=Jm +/- Jn; or JL=Jm +/- Jn; ⁵	Ureg= (CB) [Jm2 +/-Jn2] ;	JB / JL belongs to JM2	5	PD
IALU register load ⁵ JB / JL=[]; or JB / JL=Ureg; or JB / JL=<imm>	Js=Jm2 +/- Jn2 (CB);	JB / JL belongs to JM2	5	PD
IALU register load ⁵ JB / JL=[]; or JB / JL=Ureg; or JB / JL=<imm>	Ureg= (CB) [Jm2 +/-Jn2] ;	JB / JL belongs to JM2	5	PD
IALU register load ⁵ Js1=[]; or Js1=Ureg; or Js1=<imm>	Any IALU instruction ² Js=F (Jm2, Jn2);	Jm2=Js1 or Jn2=Js1	4+miss ²	PD,A
J31=Ureg; or J31=<imm>; or J31=[]; ⁵	Js=Jm+Jn+Jc; or Js=Jm - Jn+Jc - 1;	Always	5+miss ²	PD,A
J31=Ureg; or J31=<imm>; or J31=[]; ⁵	Conditional on J (jeq, jlt, jle) if <cond_j>, jump ; if <cond_j>; do ;	Always	5+miss ²	PD,A
Ureg1=[external mem- ory];; or Ureg1=SOC-Ureg	Any instruction that uses Ureg1		External access delay	PD
<ep register>=Ureg;;	Ureg=EP register	This is a bus delay that may translate to a stall. See <i>ADSP-TS201 TigerSHARC Pro- cessor Hardware Reference</i>		
SFREG=Ureg; or SFREG=[]; SFREG=<imm>;	If gsf0/1 ;	Always	4+miss	PD
SFREG=Ureg; or SFREG=[]; SFREG=<imm>;	GSF0/1 =/+=<cond>;	Always	4+miss	PD

Table 8-1. Instruction Dependencies - Stall List (Cont'd)

First Instruction	Second Instruction	Dependency	Stalls	Stage
SFREG=Ureg; or SFREG=[]; SFREG=<imm>;	X/YSF0/1 =/+<cond>;	Always	1+miss	PD
Compute instruction	GSF0/1=<compute cond>; or GSF0/1+=F(compute cond);	Compute cond of inst2 is of the same unit of inst 1	4	
X/Y/GSF0/1 =/+<cond>	[address]=SFREG; or Ureg=SFREG;		1	PD
Compute multiplier instruction or	If <multiplier condition>; X/YSF0/1=<multiplier condition>; X/YSF0/1+=op (<multiplier condition>);	Always	1	
SFREG=Ureg; or SFREG=[]; SFREG=<imm>;	If <multiplier condition>; X/YSF0/1=<multiplier condition>; X/YSF0/1+=op (<multiplier condition>);	Always	1+miss	
X/YSF0/1=<compute cond>; or X/YSF0/1+=F(compute cond);	GSF0/1=X/YSF0/1; or GSF0/1+=F(X/YSF0/1);	Same X/YSF in both instructions	4+miss	PD
LC0/1=[]; or LC0/1=Ureg	If LC0/1	Same LC	5+miss	PD
IF LC0/1E	[address]=LC0/1; or Ureg=LC0/1;	Same LC	1	PD
BTB control instructions – BTBEN; BTBDIS; BTBLOCK; BTBELOCK; BTBINV	Any instruction	Always	10	PD

Instruction Pipeline Operations

Table 8-1. Instruction Dependencies - Stall List (Cont'd)

First Instruction	Second Instruction	Dependency	Stalls	Stage
SQCTL=[]; or SQCTL=Ureg; or SQCTL=<imm>	Any instruction	Always	10	PD
<ANY DEBUG REG>=[]; or <ANY DEBUG REG>=Ureg; or <ANY DEBUG REG>=<imm>	Any instruction	Always	10	PD
Ureg=[external memory]; or Ureg=[SOC Ureg];	If <cond> RTI;	Within a level triggered interrupt ⁶	External access delay	PD
Memory accesses	See <i>ADSP-TS201 TigerSHARC Processor Hardware Reference</i>			

- 1 The stall occurs no matter if there is a data dependency or not (for design simplification)
- 2 Miss penalty by cache is added to the stall at access stage. If there is a hit, the dependency penalty is 1 cycle for compute load dependency and 4 cycles for IALU load dependency
- 3 SOC_Ureg is a Ureg in groups 0x20 to 0x3F - any Ureg in the SOC bus modules
- 4 SOC delay is the delay of the access to the external world
- 5 All the IALU rules refer to KALU as well. The example is always given for JALU
- 6 The purpose of the stall in a level interrupt is to prevent situation that the interrupt indication that is cleared by the read (e.g. read the link receive buffer that clears the buffer) is not yet cleared when exiting the interrupt routine, and this may cause a false new interrupt indication.

Stall From Compute Block Dependency

This is the most common dependency in applications and occurs on compute block operations on the compute block register file. The compute block accesses the register file for operand fetch on pipe stage Access, uses the operand on EX1, and writes the result back on EX2. The delay is comprised of basically two cycles—however, a bypass transfers the result (which is written at the end of pipe stage EX2) directly into the compute unit that is using it in the beginning of pipe stage EX1. As a result, one stall cycle is inserted in the dependent operation.

Figure 8-37 shows two compute block instructions. Execution of instruction #2 is dependent on the result from instruction #1. The pipeline inserts a one-cycle stall at the Decode stage (D) when the register dependency is recognized. When #2 reaches Decode, execution of instruction #2 is stalled for one cycle.

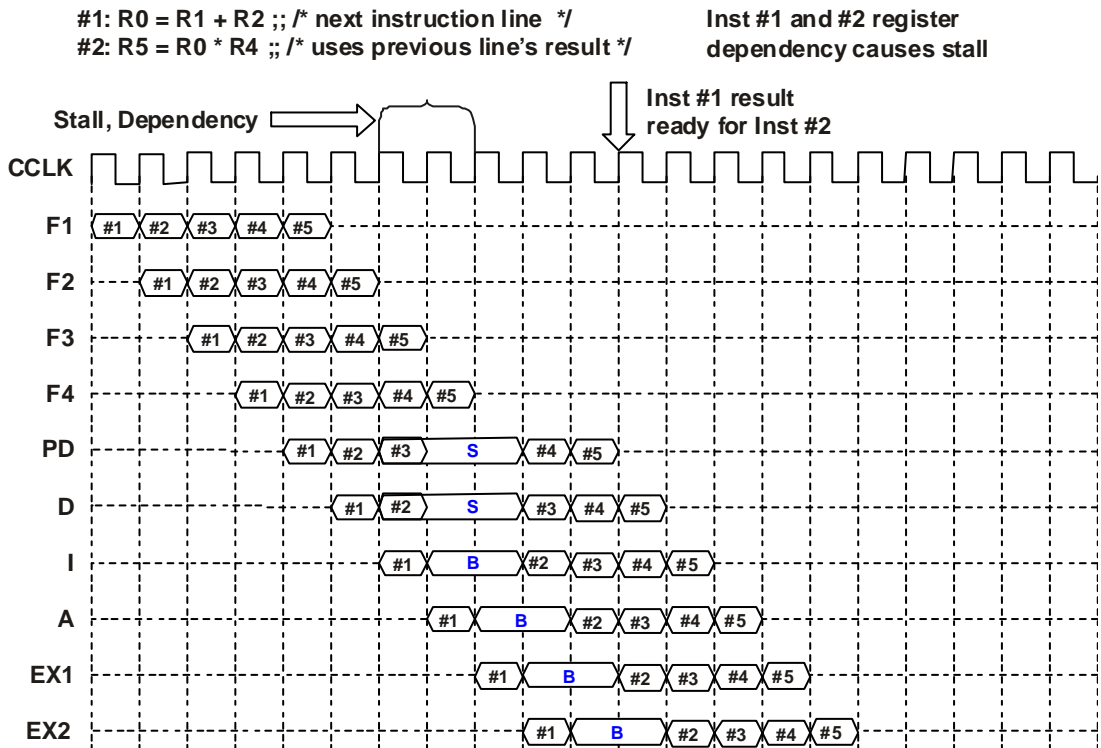


Figure 8-37. Compute Block Operations—
Result Dependency Stalls (S) and Bubbles (B) Following Instruction

As with other ALU instructions, all communications logic unit (CLU) instructions are executed in the compute pipeline. Similar to other compute instructions, all CLU instructions have a dependency check. Every use of a result of the previous line causes a stall for one cycle. In some spe-

Instruction Pipeline Operations

cial cases the stall is eliminated by using special forwarding logic in order to improve the performance of certain algorithms executions. The forwarding logic can function (and the stall can be eliminated) only when the first instruction is not predicated (for example, `if <cond>; do, <inst>;`). The exceptions cases are:

- Load `TR` or `THR` register and any instruction that uses it on the next line.
- Although the `THR` register is a hidden operand and/or result of the instructions `ACS` and `DESPREAD`, there is no dependency on it.
- The instruction `ACS`, which can use previous result of `ACS` instruction as `TRmd` with no stall. For example the following sequence will cause no stall:

```
TR3:0 = ACS (...);;  
TR7:4 = ACS (TR1:0, TR5:4, R8);;
```

Or the sequence:

```
TR3:0 = ACS (...);;  
TR7:4 = ACS (TR3:2, TR5:4, R8);;
```

However, there are a few cases that cause stalls. The first case is when the dependency is on `TRN`. For example:

```
TR3:0 = ACS (...);;  
TR7:4 = ACS (TR11:10, TR1:0, R8);;
```

The second case is when two different formats are used in the two instructions. For example:

```
XTR3:0 = ACS (TR5:4, TR7:6, R1);;  
XSTR11:8 = ACS (TR15:14, TR13:12, R2);;
```

`ACS` of short operands has the identical flow.

- Data transfer from a `CLU` register to a compute register file has no dependency. The data transfer is executed in `EX2`.

The CLU register load can be executed in parallel to other CLU instructions. CLU register load code is similar to the code of a shifter instruction, while the code of the other CLU instructions is similar to the code of ALU instructions.

No exceptions are caused by the CLU instructions.

Stall from Bus Conflict

Unlike previous TigerSHARC processors, the buses on the ADSP-TS201 processor are connected to specific bus masters (J-IALU to the J-bus, K-IALU to the K-bus, and SOC interface to the S-bus) and can access any block of memory. On previous TigerSHARC processors, the buses were connected to specific memory blocks and bus masters requested bus access. This architectural difference provides more control for eliminating bus stalls relating to bus conflicts on the ADSP-TS201 processor. To eliminate conflict related stalls, note that the execution these instructions uses the following memory buses:

- $Ureg = [Jm + Jn | imm], Ureg = [Km + Kn | imm]$
For all data types and options, the first $Ureg$ = uses J-bus, and second $Ureg$ = uses K-bus.
- $[Jm + Jn | imm] = Ureg, [Km + Kn | imm] = Ureg$
For all data types and options, the first = $Ureg$ uses J-bus, and second = $Ureg$ uses K-bus.
- $Ureg = Ureg$
 $Ureg = imm$
These use the J-Bus even if both $Ureg$ are in the same register file.
- $Js = Jm + | - Jn (CJ)$
This uses the J-Bus.
- $Ks = Km + | - Kn (CJ)$
This uses the K-Bus.

Instruction Pipeline Operations

The arbitration between the masters on the bus is detailed in “Bus Arbitration Protocol” in the *ADSP-TS201 TigerSHARC Processor Hardware Reference*.

The IALU always requests the bus on pipe stage Integer. If it doesn't receive the bus, the execution of the bus transaction is delayed until the bus is granted. The rest of the line, however, including the other IALU operations (for example, post-modify of address), are continued. This is to prevent deadlock in case of two memory accesses in the same cycle to the same bus. The following instruction lines are stalled until this line can continue executing the transaction (or transactions, if more than one of the transaction's instructions are in execution).

Figure 8-38 shows a load instruction, using an IALU post-modify addressing memory access. The memory access stalls for two cycles due to a bus conflict over access to the J-bus and J-IALU bus master. Execution of instruction #1 is extended over three cycles—two for the stall and one for the access.

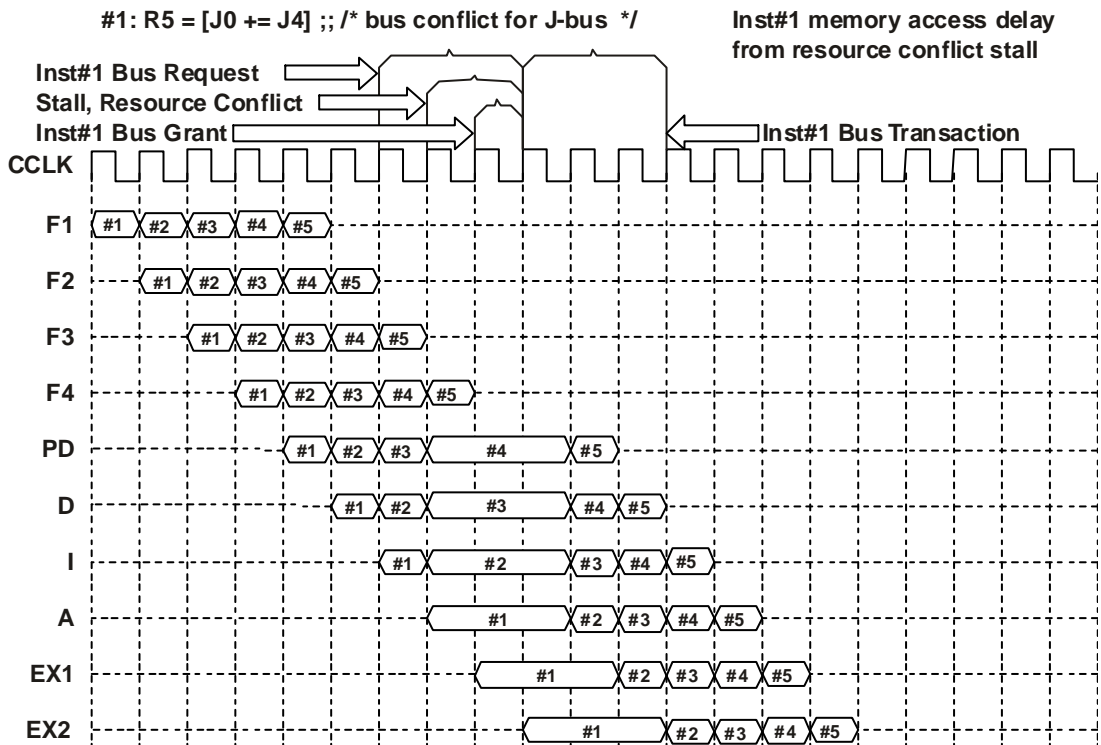


Figure 8-38. Register Load Using Post-Modify With Update—Resource Conflict Stalls Bus Access

Stall From Compute Block Load Dependency

Data in load instructions is transferred at pipe stage EX2, exactly as in compute block operations. In case of dependency between a load instruction and compute operation that uses this data, the behavior is similar to that of compute block dependency (Figure 8-39).

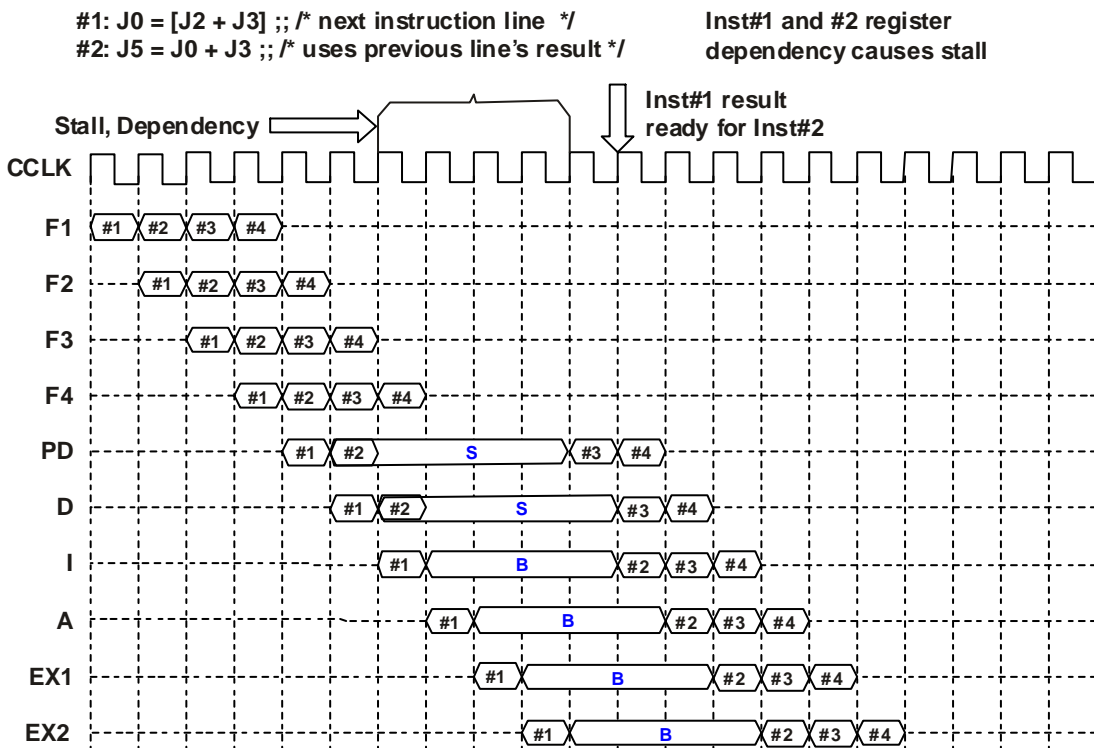


Figure 8-39. IALU Operations—
Result Dependency Stalls (S) and Bubbles (B) Following Instruction

For example, the following sequence:

```
XR0 = [J31 + memory_access] ;;
XR5 = R0 * R4 ;; /* One-cycle Stall */
```

This would cause a one-cycle delay, occurring when the load instruction comes from internal memory and the bus was accepted by the IALU that executes the transaction.

If the load is from external memory or the bus request was delayed, the second instruction is executed two cycles after the completion of the load—that is, after the data is returned.

Stall From IALU Load Dependency

The dependency between load instructions and IALU instructions is more problematic than in the previous cases because data is loaded at pipe stage EX2 and is used in stage Decode. To overcome this gap, four stalls are inserted before the instruction that is using the loaded data, as shown in [Figure 8-40 on page 8-74](#).

Stall From Load (From External Memory) Dependency

The combination of any execution instruction followed by a store instruction is dependency free, because the data is transferred by the store at pipe stage EX2. The only exception to this rule is the store of data that has been loaded from external memory. For example:

```
XR0 = [J31 + external_address] ;;
[J0+ = 0] = XR0 ;; /* stall until XR0 is ready */
```

In a case like this, there is a stall until XR0 is actually loaded.

Instruction Pipeline Operations

Stall From Conditional IALU Load Dependency

Normally IALU instructions are executed in a single cycle at pipe stage Integer. The result is pipelined and written into the result register at pipe stage EX2. If the following instruction uses the result of this instruction (either the result is used or a condition is used), the sequential instruction extracts the result from the pipeline. In one exceptional instance the bypass cannot be used, as shown in [Figure 8-40](#).

#1: IF AEQ; DO, J0 = J2 + J3 ;; /* next instruction line */
#2: J5 = J0 + J3 ;; /* uses previous line's result */

Inst#1 and #2 register
dependency causes stall

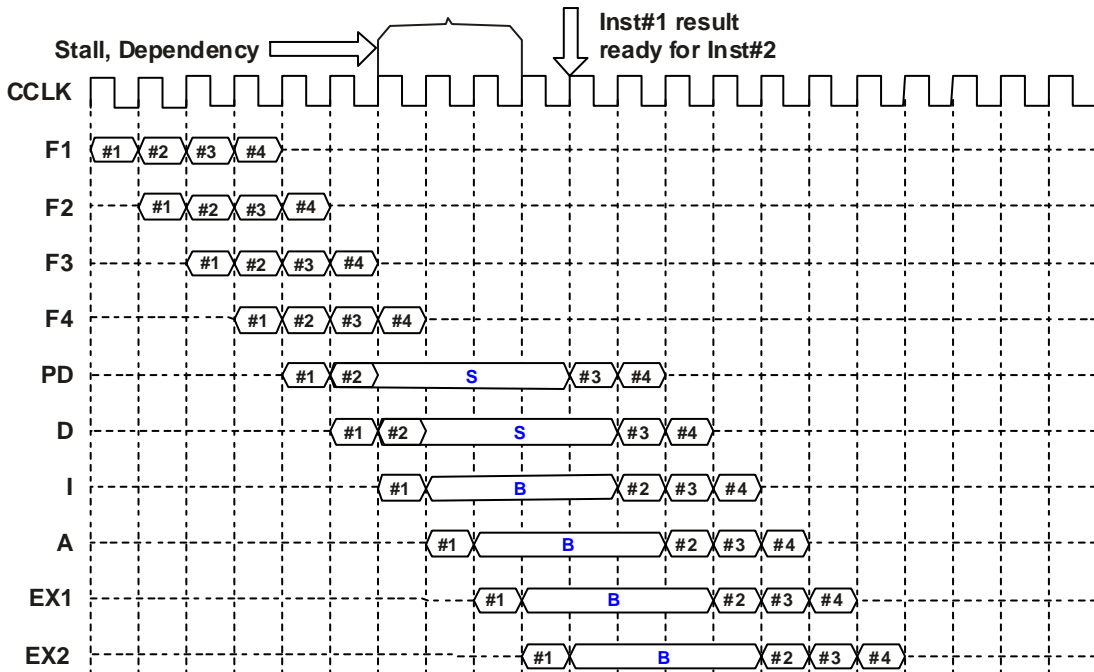


Figure 8-40. IALU Conditional Operations—
Result Dependency Stalls Following Instruction

This occurs when the first instruction is conditional, the bypass usage is conditional, and the condition value is not known yet. The result of inst#1 in the example cannot be extracted from the bypass and must be taken from the J0 register after the completion of the execution—after pipe stage EX2. In this case, three stall cycles are inserted if the condition is compute block, and one cycle is inserted for other types of conditions.

Interrupt Effects on Pipeline

Interrupts (and exceptions) break the flow of execution and cause pipeline effects similar to other types of branching execution. The interrupt types are described in the “Interrupts” chapter of the *ADSP-TS201 TigerSHARC Processor Hardware Reference*.

The interrupts in some applications are performance critical, and the ADSP-TS201 processor executes them (in most cases) in the same pipeline in the optimal flow. The next sections describe the different flows of interrupts.

The most common case of a hardware interrupt is shown in [Figure 8-41](#). When an interrupt is identified by the core (when the interrupt bit in ILAT register is set) or when the interrupt becomes enabled (the interrupt bit in IMASK register is set), the ADSP-TS201 processor starts fetching from the interrupt routine address. The execution of the instructions of the regular flow continues, except for the last instruction before the interrupt (Inst #2 in the previous example). The return address saved in RETI would be the address of instruction 2.

Instruction Pipeline Operations

#1: /* any unconditional, non-branching instruction */

#2: /* any unconditional, non-branching instruction */

#1i: /* interrupt service routine instruction #1 */

#2i: /* interrupt service routine instruction #2 */

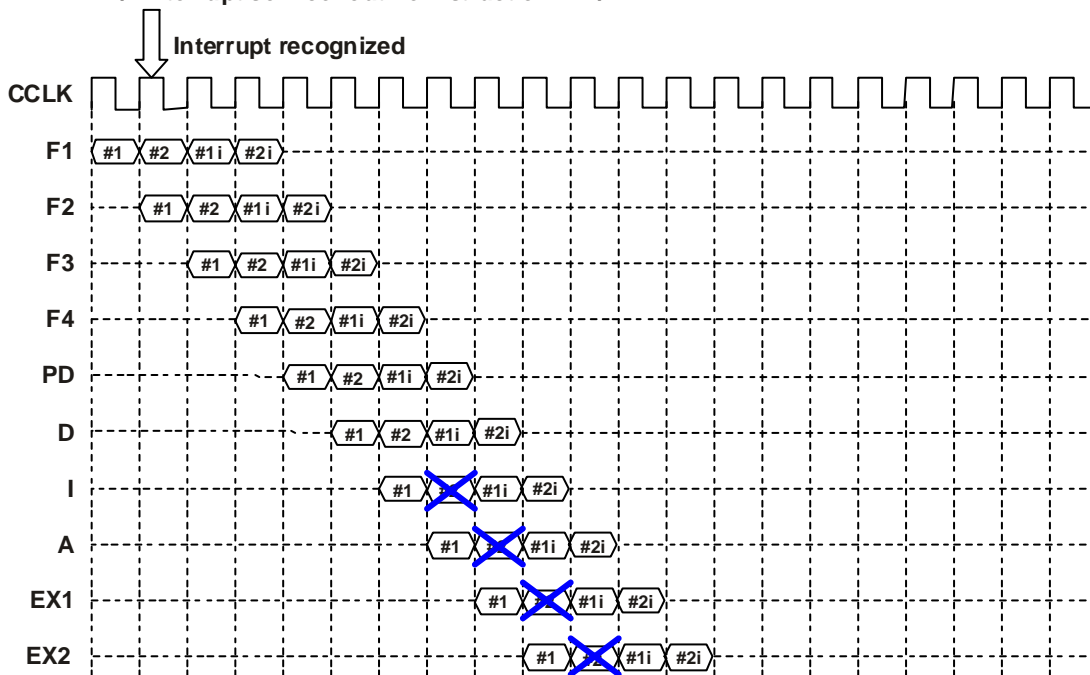


Figure 8-41. Interrupting Linear Execution (Interrupts Enabled, Interrupting Unconditional, Non-Branching Instructions)

Interrupt During Conditional Instruction

When a predicted branch or not predicted branch instruction is fetched, the ADSP-TS201 processor cannot decide immediately if the branch is to be taken (as discussed in [“Branch Target Buffer \(BTB\)” on page 8-38](#)). When an interrupt occurs during a predicted branch or not predicted branch instruction, if the prediction is found incorrect, the predicted part is aborted while the interrupt instructions that follow are not aborted.

This case is illustrated in [Figure 8-42](#). When the interrupt is inserted into the flow, instructions #3 and #4 are in the pipeline speculatively. When the jump instruction is finalized (EX2) and if the speculative is found wrong, instructions #3 and #4 are aborted (similar to the flow described in [Figure 8-43 on page 8-80](#)).

The instructions that belong to the interrupt flow, however, are not part of the speculative flow, and they are not aborted. The return address in this case is the correct target of the jump instruction #2. Similar flows happen in all cases of aborted predicted or not predicted flows, when interrupt routine instructions are already in the pipeline.

Instruction Pipeline Operations

#1: XR0 = R2 - R1 ;; /* updates flags for inst#2 */

#2: IF NXAEQ, JUMP 100 ;; /* conditional on result from inst#1

#1i: /* interrupt service routine instruction #1 */

#2i: /* interrupt service routine instruction #2 */

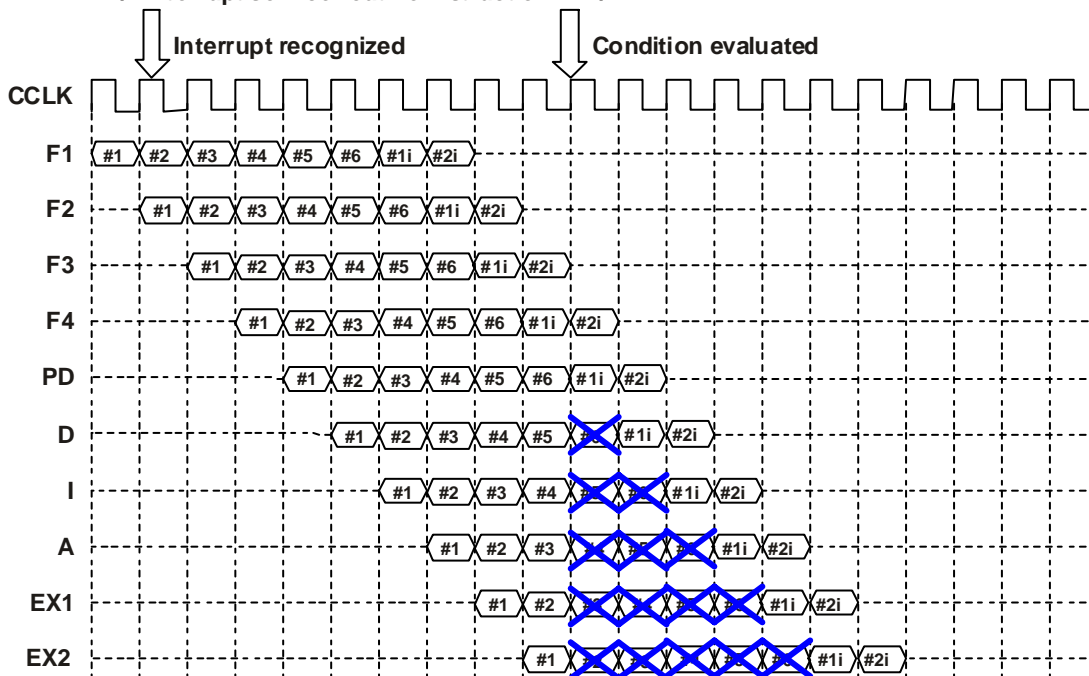


Figure 8-42. Interrupting Conditional Execution (Interrupts Enabled, Interrupting Conditional Instructions)

Interrupt During Interrupt Disable Instruction

Sometimes the programmer needs a certain part of the code to be executed free of interrupts. In this case, disabling all hardware interrupts by clearing the global interrupt enable `GIE` bit of `SQCTL` register is effective immediately (contrary to clearing a specific interrupt enable). Be aware that there is a performance cost to using this feature. If the interrupt is already in the pipeline when `GIE` is cleared, it will continue execution until reaching `EX1`, and only then will it be aborted and the flow returned to a normal flow.

An example to this flow is shown in [Figure 8-43](#). The interrupt is identified by the ADSP-TS201 processor on the second cycle (when inst #1 is fetched). Inst #1—which clears `GIE`—is only completed five cycles after the interrupt occurs. When the first interrupt routine instruction reaches `EX1`, `GIE` is checked again, and if it is cleared, the whole interrupt flow is aborted and the TigerSHARC processor returns to its original flow.

Exception Effects on Pipeline

An exception is normally caused by using a specific instruction line. The exception routine's first instruction is the next instruction executed after the instruction that caused it. In order to make this happen, when the instruction line that caused the exception reaches `EX2`, all the instructions in the pipeline are aborted, and the ADSP-TS201 processor starts fetching from the exception routine. This flow is similar to the flow of unpredicted and taken jumps conditioned by an `EX2` condition (see [Figure 8-31 on page 8-53](#)).

Instruction Pipeline Operations

```

#_: XR0 = SQCTL ;; /* load XR0 with current interrupt mask */
#_: XR0 = BCLR R0 BY INT_GIE;;
    /* where INT_GIE = 0xFFFFFBB from defs201.h file;
       this clears the GIE bit, but retains other bit values */
#1: SQCTL = XR0 ;; /* load SQCTL with data globally disabling interrupts */
#2: /* any unconditional instruction */

```

```
#1i: /* interrupt service routine instruction #1 */
```

```
#2i: /* interrupt service routine instruction #2 */
```

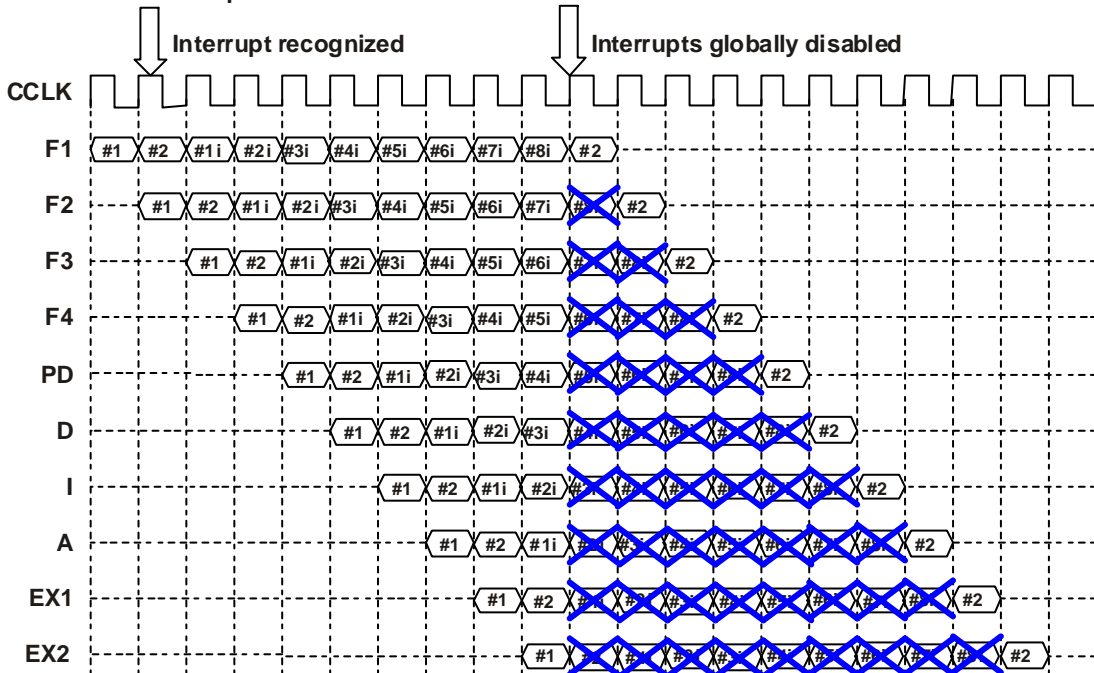


Figure 8-43. Interrupting Linear Execution (Interrupts Enabled, Interrupting Unconditional, Non-Branching Instructions)—Interrupt Disabled While In Pipeline

Sequencer Examples

The listings in this section provide examples of sequencer instruction usage. The comments with the instructions identify the key features of the instruction, such as predicted or not predicted branches, loop setup, and others.

Listing 8-5. Sequencer Instructions, Slots, and Lines

```
IF XAEQ, JUMP Label_1NP;;
/* This is a single instruction line and occupies one "slot" or
32-bit word */
NOP; NOP; NOP; NOP;;
IF XMEQ; D0, XR3:2 = R5:4 + R7:6;;
/* This is a two instruction line, the first slot is the condi-
tional instruction XMEQ, the second instruction is the addition
in the X computation block */
```

Listing 8-6. JUMP Instruction Example

```
.SECTION program ;
CALL test ;; /* Addr:0x0; */
/* Jumps to 0x3. Stores 0x1 in CJMP register */
NOP ;; /* Addr: 0x1 */
endhere:
    JUMP endhere ;; /* Addr: 0x2 */
test:
    NOP ;; /* Addr: 0x3 */
    CJMP (ABS) ;; /* Addr: 0x4; */
    /* End of subroutine. Jumps back to Addr 0x1. */
```

Sequencer Examples

Listing 8-7. CJMP_CALL Instruction Example

```
.SECTION program;
J0 = ADDRESS(endhere) ;; /* Addr:0x0 */
CJMP = ADDRESS(test) ;; /* Addr: 0x1 */
/* Preload cjmp register with 0x6 */
CJMP_CALL (ABS) ;; /* 0x2 */
/* Jumps to value stored in CJMP register (0x6).
/* Puts new value of 0x3 in CJMP register. */
NOP ;; /* Addr: 0x3 */
endhere:
    NOP ;; /* Addr: 0x4 */
    JUMP endhere ;; /* Addr: 0x5 */
test:
    J31 = J0 + J31 (CJMP) ;; /* Addr: 0x6 */
    /* Uses IALU add (CJMP) option */
    /* Loads result (0x4) into CJMP register. */
    CJMP (ABS) ;; /* Addr: 0x7 */
    /* End of subroutine. Jumps back to 0x4. */
```

Listing 8-8. Zero-Overhead and Near-Zero-Overhead Loops Example

```
J6 = J31 + 10;; /*initialize counter for outermost loop */
Outer_loop:
    instr0;;
    LC1 = 5; /* Counter for middle loop. */
    /* Use zero overhead in loop counter number one */
    instr1;;
    Middle_loop:
        instr2;;
        LC0 = 6; instr3;; /* Counter for inner loop. */
        /* Use loop counter zero */
        Inner_loop:
            instr4;;
```

```
        instr5;;
    If NLC0E, JUMP Inner_loop; instr6;; /* End inner loop */
    instr7;;
    If NLC1E, JUMP Middle_loop; instr8;; /* End middle loop */
    instr9; J6 = J6 - 1;; /* Decrement counter for outer loop */
    instr10;;
    If NJEQ, JUMP Outer_loop; instr11;; /* End outer loop*/
```

Listing 8-9. Branch Target Buffer (BTB) Usage (No Branch Prediction)

```
LC0 = 100;; /* Initialize loop count */
instr2;;
instr3;;
_loop:
    instr4;;
    instr5;;
    If NLC0E, JUMP _loop (NP); instr6;; /* End loop */
```

Some notes on [Listing 8-9](#):

- “NP” denotes no predict
- First 99 times through the loop there is a three cycle penalty
- Last time though the loop there is no penalty
- Total penalty cycles => 297

Listing 8-10. Branch Target Buffer (BTB) Usage (Branch Prediction)

```
SQCTL=0xF0301;; /* Make sure BTB is enabled */
LC0 = 100;; /* Initialize loop count */
instr2;;
instr3;;
_loop:
    instr4;;
```

Sequencer Instruction Summary

```
instr5;;  
If NLC0E, JUMP _loop; instr6;; /* End loop */
```

Some notes on [Listing 8-10](#):

- First time through (and second time through if the branch is not updated in the BTB) the loop there is a two cycle penalty
- Next 98 times through the loop there is no penalty
- Last time through the loop there is a three cycle penalty
- Total penalty cycles => five (seven if no BTB update)

Sequencer Instruction Summary

[Listing 8-11](#) shows the sequencer instructions' syntax. The conventions used in these listings for representing register names, optional items, and choices are covered in detail in [“Register File Registers” on page 2-5](#).

Briefly, these conventions are:

- { } – the curly braces enclose options; these braces are not part of the instruction syntax.
- | – the vertical bars separate choices; these bars are not part of the instruction syntax.
- *Label* – the program label in italic represents a user-selectable program label, a PC-relative 16- or 32-bit address, or a 32-bit absolute address. When a program *Label* is used instead of an address, the assembler converts the *Label* to an address, using a 16-bit address when the *Label* is contained in the same program `.SECTION` as the branch instruction and using a 32-bit address when the *Label* is

not in the same program `.SECTION` as the branch instruction. For more information on relative and absolute addresses and branching, see “[Branching Execution](#)” on page 8-16.



Each instruction presented here occupies one instruction slot in an instruction line. For more information about instruction lines and instruction combination constraints, see “[Instruction Line Syntax and Structure](#)” on page 1-22 and “[Instruction Parallelism Rules](#)” on page 1-26.

PC-relative addressing and predicted branch for jumps and calls are the default operation. To use absolute addressing, use the `(ABS)` option. To indicate a not predicted branch, use the `(NP)` option.

In conditional instructions, the *Condition* is one of the following:

- `{N}{X|Y|XY} AEQ|ALT|ALE|MEQ|MLT|MLE|SEQ|SLT|SFO|SF1`
- `{N}{J|K} EQ|LT|LE`
- `{N}ISF0|ISF1|BM|LC0E|LC1E|FLG0|FLG1|FLG2|FLG3.`

The `AEQ`, `ALT`, and `ALE` conditions are the ALU equal, less than, and less than or equal to zero conditions. The `MEQ`, `MLT`, and `MLE` conditions are the multiplier equal, less than, and less than or equal to zero conditions. The `SEQ` and `SLT` conditions are the Shifter equal and less than zero conditions. `SFO` and `SF1` are the compute block static flag 0 and 1 conditions.

The `JEQ`, `JLT`, and `JLE` conditions are the J-ALU equal, less than, and less than or equal to zero conditions and, similarly, for K-ALU.

The `ISF0` and `ISF1` conditions are the general integer static flag conditions. `BM` is bus master. The `LC0E` and `LC1E` conditions are the loop counter 0 and 1 equal to zero conditions. The `FLG0` through `FLG3` conditions are the flag pin conditions.

Sequencer Instruction Summary

An **N** prefix on a condition negates the condition in the instruction. Thus, **NJLT** is “J-IALU greater than or equal to zero.” For example, **N** in the following instruction

```
IF NXMLT, JUMP label_10;;
```

is used to execute a jump to the address that corresponds to `label_10` if the condition **MLT** (multiplier less than zero) in compute block **X** evaluates false.

JUMP|...|CJMP denotes any valid branch instruction. This is a line with more than one instruction.

Instructions that follow after a conditional instruction or after a jump/call in a line can have keyword **DO|ELSE** or have no prefix. Absence of prefix indicates execution regardless of the condition value. Keyword **DO** relates only to condition instruction, while **ELSE** relates only to conditional branch.



The **NOP**, **IDLE** (`lp`), **BTBINV**, **TRAP** (`<imm>`), and **EMUTRAP** instructions may not be conditional. The following instruction line for example is not legal: `if aeq; do idle;;`

Listing 8-11. Sequencer Instructions

```
{IF Condition,} JUMP|CALL <Label> {(NP)} {(ABS)} ;
```

```
{IF Condition,} CJMP|CJMP_CALL {(NP)} {(ABS)} ;
```

```
{IF Condition,} RETI|RTI {(NP)} {(ABS)} ;
```

```
{IF Condition,} RDS ;
```

```
IF Condition;
```

```
DO, instruction; DO, instruction; DO, instruction ;;
```

```
/* This syntax permits up to three instructions to be controlled  
by a condition. Omitting the D0 before the instruction makes the  
instruction unconditional. */
```

```
IF Condition, JUMP|CALL|CJMP|CJMP_CALL ;  
    ELSE, instruction; ELSE, instruction; ELSE, instruction ;;  
/* This syntax permits up to three instructions to be controlled  
by a condition. Omitting the ELSE before the instruction makes  
the instruction unconditional. */
```

```
SF1|SF0 = Condition ;
```

```
SF1|SF0 += AND|OR|XOR Condition ;
```

```
IDLE ;
```

```
BTBINV
```

```
TRAP (<Imm5>) ;;
```

```
EMUTRAP ;;
```

```
NOP ;
```

Sequencer Instruction Summary