Program Sequencing 3

3.1 OVERVIEW

Program flow in the ADSP-2106x is most often linear; the processor executes program instructions sequentially. Variations in this linear flow are provided by the following program structures, illustrated in Figure 3.1 on the following page:

- *Loops.* One sequence of instructions is executed several times with zero overhead.
- *Subroutines.* The processor temporarily interrupts sequential flow to execute instructions from another part of program memory.
- *Jumps.* Program flow is permanently transferred to another part of program memory.
- *Interrupts.* A special case of subroutines in which the execution of the routine is triggered by an event that happens at run time, not by a program instruction.
- *Idle.* A special instruction that causes the processor to cease operations, holding its current state. When an interrupt occurs, the processor services the interrupt and continues normal execution.

Managing these program structures is the job of the ADSP-2106x's program sequencer. The program sequencer selects the address of the next instruction, generating most of those addresses itself. It also performs a wide range of related functions, such as

- incrementing the fetch address,
- maintaining stacks,
- evaluating conditions,
- decrementing the loop counter,
- calculating new addresses,
- maintaining an instruction cache, and
- handling interrupts.

3 Program Sequencing

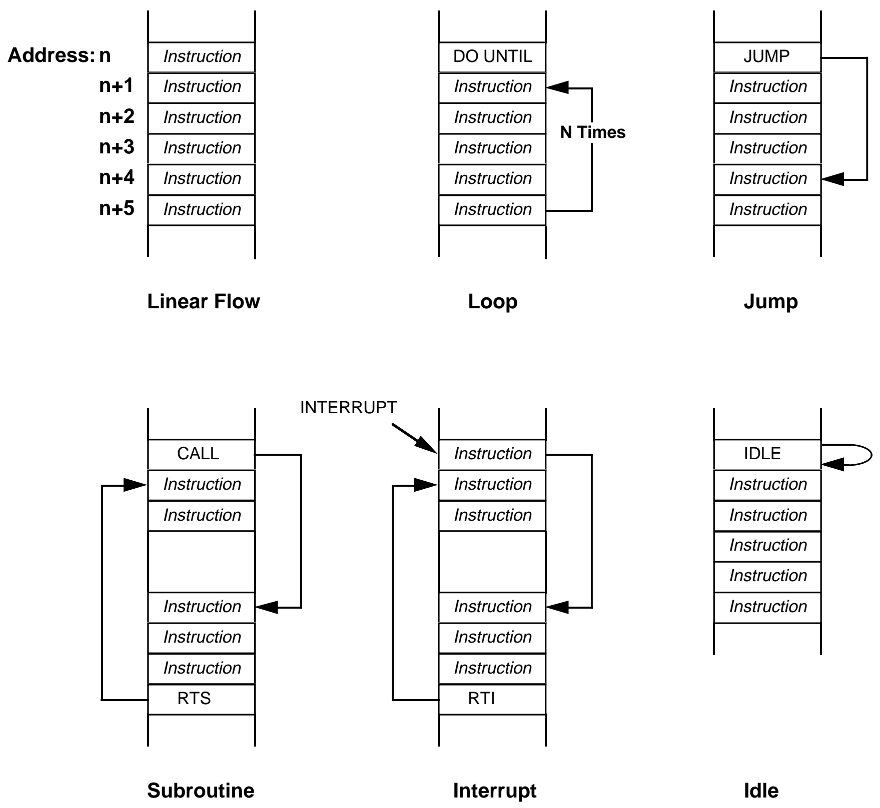


Figure 3.1 Program Flow Variations

3.1.1 Instruction Cycle

The ADSP-2106x processes instructions in three clock cycles:

- In the *fetch* cycle, the ADSP-2106x reads the instruction from either the on-chip instruction cache or from program memory.
- During the *decode* cycle, the instruction is decoded, generating conditions that control instruction execution.
- In the *execute* cycle, the ADSP-2106x executes the instruction; the operations specified by the instruction are completed.

These cycles are overlapping, or pipelined, as shown in Figure 3.2. In sequential program flow, when one instruction is being fetched, the instruction fetched in the previous cycle is being decoded, and the instruction fetched two cycles before is being executed. Thus, the throughput is one instruction per cycle.

time (cycles)	Fetch	Decode	Execute
1	0x08		
2	0x09	0x08	
3	0x0A	0x09	0x08
4	0x0B	0x0A	0x09
5	0x0C	0x0B	0x0A

Figure 3.2 Pipelined Execution Cycles

Any non-sequential program flow can potentially decrease the ADSP-2106x's instruction throughput. Non-sequential program operations include:

- Program memory data accesses that conflict with instruction fetches
- Jumps
- Subroutine Calls and Returns
- Interrupts and Returns
- Loops

3.1.2 Program Sequencer Architecture

Figure 3.3, on the next page, shows a block diagram of the program sequencer. The sequencer selects the value of the next fetch address from several possible sources.

The fetch address register, decode address register and program counter (PC) contain, respectively, the addresses of the instructions currently being fetched, decoded and executed. The PC is coupled with the PC stack, which is used to store return addresses and top-of-loop addresses.

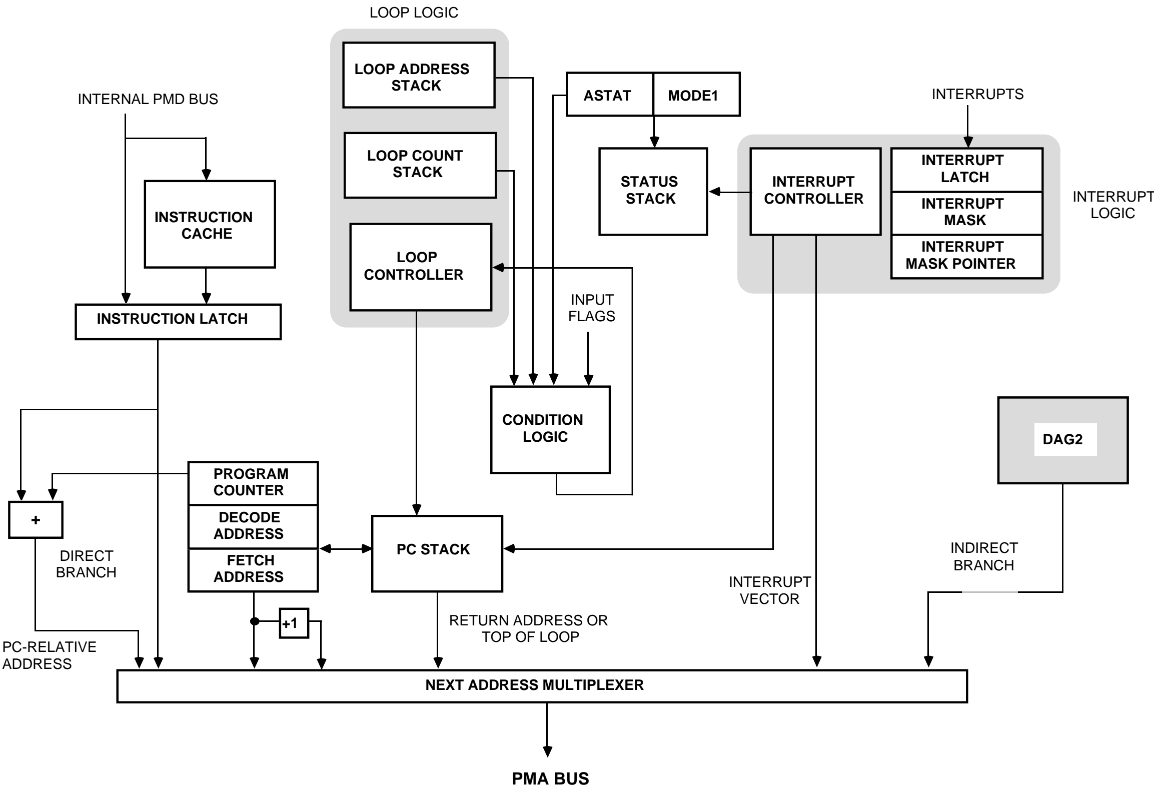


Figure 3.3 Program Sequencer Block Diagram

The interrupt controller performs all functions related to interrupt processing, such as determining whether an interrupt is masked and generating the appropriate interrupt vector address.

The instruction cache provides the means by which the ADSP-2106x can access data in program memory and fetch an instruction (from the cache) in the same cycle. The DAG2 data address generator (described in the next chapter) outputs program memory data addresses.

The sequencer evaluates conditional instructions and loop termination conditions using information from the status registers. The loop address stack and loop counter stack support nested loops. The status stack stores status registers for implementing nested interrupt routines.

3.1.2.1 Program Sequencer Registers & System Registers

Table 3.1 lists the registers located in the program sequencer. The functions of these registers are described in subsequent sections of this chapter. All registers in the program sequencer are universal registers and are thus accessible to other universal registers as well as to data memory. All registers and the tops of stacks are readable; all registers except the fetch address, decode address and PC are writeable. The PC stack can be pushed and popped by writing the PC stack pointer, which is readable and writeable. The loop address stack and status stack are pushed and popped by explicit instructions.

The *System Register Bit Manipulation* instruction can be used to set, clear, toggle or test specific bits in the system registers. This instruction is described in Appendix A, Group IV–Miscellaneous Instructions.

Due to pipelining, writes to some of these registers do not take effect on the next cycle; for example, if you write the MODE1 register to enable ALU saturation mode, the change will not occur until two cycles after the write. Also, some registers are not updated on the cycle immediately following a write; it takes an extra cycle before a read of the register yields the new value. Table 3.1 summarizes the number of extra cycles for a write to take effect (*effect latency*) and for a new value to appear in the register (*read latency*). A “0” indicates that the write takes effect or appears in the register on the next cycle after the write instruction is executed. A “1” indicates one extra cycle.

Program Sequencer Registers	*Contents*	*Bits*	*Read Latency*	*Effect Latency*
FADDR*	fetch address	24	–	–
DADDR*	decode address	24	–	–
PC*	execute address	24	–	–
PCSTK	top of PC stack	24	0	0
PCSTKP	PC stack pointer	5	1	1
LADDR	top of loop address stack	32	0	0
CURLCNTR	top of loop count stack (current loop count)	32	0	0
LCNTR	loop count for next DO UNTIL loop	32	0	0
System Registers				
MODE1	mode control bits	32	0	1
MODE2	mode control bits	32	0	1
IRPTL	interrupt latch	32	0	1
IMASK	interrupt mask	32	0	1
IMASKP	interrupt mask pointer (for nesting)	32	1	1
ASTAT	arithmetic status flags	32	0	1
STKY	sticky status flags	32	0	1
USTAT1	user-defined status flags	32	0	0
USTAT2	user-defined status flags	32	0	0

Table 3.1 Program Sequencer Registers & System Registers

* read-only

3.2 PROGRAM SEQUENCER OPERATIONS

This section gives an overview of the operation of the program sequencer. The various kinds of program flow are defined here and described in detail in subsequent sections.

3.2.1 Sequential Instruction Flow

The program sequencer determines the next instruction address by examining both the current instruction being executed and the current state of the processor. If no conditions require otherwise, the ADSP-2106x executes instructions from program memory in sequential order by simply incrementing the fetch address.

3.2.2 Program Memory Data Accesses

Usually, the ADSP-2106x fetches an instruction from memory on each cycle. When the ADSP-2106x executes an instruction which requires data to be read from or written to the same memory block in which the instruction is stored, there is a conflict for access to that block. The ADSP-2106x uses its instruction cache to reduce delays caused by this type of conflict.

The first time the ADSP-2106x encounters an instruction fetch that conflicts with a program memory data access, it must wait to fetch the instruction on the following cycle, causing a delay. The ADSP-2106x automatically writes the fetched instruction to the cache to prevent the same delay from happening again. The ADSP-2106x checks the instruction cache on every program memory data access. If the instruction needed is in the cache, the instruction fetch from the cache happens in parallel with the program memory data access, without incurring a delay.

3.2.3 Branches

A branch occurs when the fetch address is not the next sequential address following the previous fetch address. Jumps, calls and returns are the types of branches which the ADSP-2106x supports. In the program sequencer, the only difference between a jump and a call is that upon execution of a call, a return address is pushed onto the PC stack so that it is available when a return instruction is later executed. Jumps branch to a new location without allowing return.

3.2.4 Loops

The ADSP-2106x supports program loops with the DO UNTIL instruction. The DO UNTIL instruction causes the ADSP-2106x to repeat a sequence of instructions until a specified condition tests true.

3.3 CONDITIONAL INSTRUCTION EXECUTION

The program sequencer evaluates conditions to determine whether to execute a conditional instruction and when to terminate a loop. The conditions are based on information from the arithmetic status (ASTAT) register, mode control 1 (MODE1) register, flag inputs and loop counter. The arithmetic ASTAT bits are described in the previous chapter, *Computation Units*.

Each condition that the ADSP-2106x evaluates has an assembler mnemonic and a unique code which is used in a conditional instruction's opcode. For most conditions, the program sequencer can test both true and false states, e.g., equal to zero and not equal to zero. Table 3.2, on the following page, defines the 32 condition and termination codes.

The bit test flag (BTF) is bit 18 of the ASTAT register. This flag is set (or cleared) by the results of the BIT TST and BIT XOR forms of the *System Register Bit Manipulation* instruction, which can be used to test the contents of the ADSP-2106x's system registers. This instruction is described in Appendix A, Group IV–Miscellaneous instructions. After BTF is set by this instruction, it can be used as the condition in a conditional instruction (with the mnemonic TF; see Table 3.2).

The two conditions that do not have complements are LCE/NOT LCE (loop counter expired/not expired) and TRUE/FOREVER. The interpretation of these condition codes is determined by context; TRUE and NOT LCE are used in conditional instructions, FOREVER and LCE in loop termination. The IF TRUE construct creates an unconditional instruction (the same effect as leaving out the condition entirely). A DO FOREVER instruction executes a loop indefinitely, until an interrupt or reset intervenes.

The LCE condition (loop counter expired) is most commonly used in a DO UNTIL instruction. Because the LCE condition checks the value of the loop counter (CURLCNTR), an IF NOT LCE conditional instruction should not follow a write to CURLCNTR from memory. Otherwise, because the write occurs after the NOT LCE test, the condition is based on the old CURLCNTR value.

The bus master condition (BM) indicates whether the ADSP-2106x is the current bus master in a multiprocessor system. To enable the use of this condition, bits 17 and 18 of the MODE1 register must both be zeros; otherwise the condition is always evaluated as false.

No.	Mnemonic	Description	True If
0	EQ	ALU equal zero	AZ = 1
1	LT	ALU less than zero	*See Note 1 below*
2	LE	ALU less than or equal zero	*See Note 2 below*
3	AC	ALU carry	AC = 1
4	AV	ALU overflow	AV = 1
5	MV	Multiplier overflow	MV = 1
6	MS	Multiplier sign	MN = 1
7	SV	Shifter overflow	SV = 1
8	SZ	Shifter zero	SZ = 1
9	FLAG0_IN	Flag 0 input	FI0 = 1
10	FLAG1_IN	Flag 1 input	FI1 = 1
11	FLAG2_IN	Flag 2 input	FI2 = 1
12	FLAG3_IN	Flag 3 input	FI3 = 1
13	TF	Bit test flag	BTF = 1
14	BM	Bus Master	
15	LCE	Loop counter expired (DO UNTIL term)	CURLCNTR = 1
15	NOT LCE	Loop counter not expired (IF cond)	CURLCNTR ≠ 1
Bits 16-30 are the complements of bits 0-14			
16	NE	ALU not equal to zero	AZ = 0
17	GE	ALU greater than or equal zero	*See Note 3 below*
18	GT	ALU greater than zero	*See Note 4 below*
19	NOT AC	Not ALU carry	AC = 0
20	NOT AV	Not ALU overflow	AV = 0
21	NOT MV	Not multiplier overflow	MV = 0
22	NOT MS	Not multiplier sign	MN = 0
23	NOT SV	Not shifter overflow	SV = 0
24	NOT SZ	Not shifter zero	SZ = 0
25	NOT FLAG0_IN	Not Flag 0 input	FI0 = 0
26	NOT FLAG1_IN	Not Flag 1 input	FI1 = 0
27	NOT FLAG2_IN	Not Flag 2 input	FI2 = 0
28	NOT FLAG3_IN	Not Flag 3 input	FI3 = 0
29	NOT TF	Not bit test flag	BTF = 0
30	NBM	Not Bus Master	
31	FOREVER	Always False (DO UNTIL)	always
31	TRUE	Always True (IF)	always

Table 3.2 Condition & Loop Termination Codes

Notes:
1. [$\overline{AF}$ and (AN xor (AV and $\overline{ALUSAT}$)) or (AF and AN and $\overline{AZ}$)] = 1
2. [$\overline{AF}$ and (AN xor (AV and $\overline{ALUSAT}$)) or (AF and AN)] or AZ = 1
3. [$\overline{AF}$ and (AN xor (AV and $\overline{ALUSAT}$)) or (AF and AN and $\overline{AZ}$)] = 0
4. [$\overline{AF}$ and (AN xor (AV and $\overline{ALUSAT}$)) or (AF and AN)] or AZ = 0

3.4 BRANCHES (CALL, JUMP, RTS, RTI)

The CALL instruction initiates a subroutine. Both jumps and calls transfer program flow to another memory location, but a call also pushes a return address onto the PC stack so that it is available when a return from subroutine instruction is later executed. Jumps branch to a new location without allowing return.

A return causes the processor to branch to the address stored at the top of the PC stack. There are two types of returns: return from subroutine (RTS) and return from interrupt (RTI). The difference between the two is that the RTI instruction not only pops the return address off the PC stack, but also: 1) pops the status stack if the ASTAT and MODE1 status registers have been pushed (if the interrupt was $\overline{IRQ}_{2\text{-}0}$, the timer interrupt, or the VIRPT vector interrupt), and 2) clears the appropriate bit in the interrupt latch register (IRPTL) and the interrupt mask pointer (IMASKP).

There are a number of parameters you can specify for branches:

- Jumps, calls and returns can be conditional. The program sequencer can evaluate any one of several status conditions to decide whether the branch should be taken. If no condition is specified, the branch is always taken.
- Jumps and calls can be indirect, direct, or PC-relative. An *indirect* branch goes to an address supplied by one of the data address generators, DAG2. *Direct* branches jump to the 24-bit address specified in an immediate field in the branch instruction. *PC-relative* branches also use a value specified in the instruction, but the sequencer adds this value to the current PC value to compute the destination address.
- Jumps, calls and returns can be delayed or nondelayed. In a *delayed* branch, the two instructions immediately after the branch instruction are executed; in a *nondelayed* branch, the program sequencer suppresses the execution of those two instructions (NOPs are performed instead).
- The JUMP (LA) instruction causes an automatic loop abort if it occurs inside a loop. When the loop is aborted, the PC and loop address stacks are popped once, so that if the loop was nested, the stacks still contain the correct values for the outer loop. JUMP (LA) is similar to the *break* instruction of the C programming language used to prematurely terminate execution of a loop. (Note: JUMP (LA) may not be used in the last three instructions of a loop.)

3.4.1 Delayed & Nondelayed Branches

An instruction modifier (DB) indicates that a branch is delayed; otherwise, it is nondelayed. If the branch is nondelayed, the two instructions after the branch, which are in the fetch and decode stages, are not executed (see Figure 3.4); for a call, the decode address (the address of the instruction after the call) is the return address. During the two no-operation cycles, the first instruction at the branch address is fetched and decoded.

NON-DELAYED JUMP OR CALL

CLOCK CYCLES →

Execute Instruction	n	nop	nop	j
Decode Instruction	n+1->nop	n+2->nop	j	j+1
Fetch Instruction	n+2	j	j+1	j+2
	n+1 suppressed	n+2 suppressed; for call, n+1 pushed on PC stack		

NON-DELAYED RETURN

CLOCK CYCLES →

Execute Instruction	n	nop	nop	r
Decode Instruction	n+1->nop	n+2->nop	r	r+1
Fetch Instruction	n+2	r	r+1	r+2
	n+1 suppressed	n+2 suppressed; r popped from PC stack		

n = Branch instruction

j = Instruction at Jump or Call address

r = Instruction at Return address

Figure 3.4 Nondelayed Branches

In a delayed branch, the processor continues to execute two more instructions while the instruction at the branch address is fetched and decoded (see Figure 3.5); in the case of a call, the return address is the third address after the branch instruction. A delayed branch is more efficient, but it makes the code harder to understand because of the instructions between the branch instruction and the actual branch.

DELAYED JUMP OR CALL

CLOCK CYCLES →

Execute Instruction	n	n+1	n+2	j
Decode Instruction	n+1	n+2	j	j+1
Fetch Instruction	n+2	j	j+1	j+2

for call, n+3 pushed on PC stack

DELAYED RETURN

CLOCK CYCLES →

Execute Instruction	n	n+1	n+2	r
Decode Instruction	n+1	n+2	r	r+1
Fetch Instruction	n+2	r	r+1	r+2

r popped from PC stack

n = Branch instruction

j = Instruction at Jump or Call address

r = Instruction at Return address

Figure 3.5 Delayed Branches

Because of the instruction pipeline, a delayed branch instruction and the two instructions that follow it must be executed sequentially. Instructions in the two locations immediately following a delayed branch instruction may not be any of the following:

- Other Jumps, Calls or Returns
- Pushes or Pops of the PC stack
- Writes to the PC stack or PC stack pointer
- DO UNTIL instruction
- IDLE or IDLE16 instruction

These exceptions are checked by the ADSP-21000 Family assembler.

The ADSP-2106x does not process an interrupt in between a delayed branch instruction and either of the two instructions that follow, since these three instructions must be executed sequentially. Any interrupt that occurs during these instructions is latched but not processed until the branch is complete.

A read of the PC stack or PC stack pointer immediately after a delayed call or return is permitted, but it will show that the return address on the PC stack has already been pushed or popped, even though the branch has not occurred yet.

3.4.2 PC Stack

The PC stack holds return addresses for subroutines and interrupt service routines and top-of-loop addresses for loops. The PC stack is 30 locations deep by 24 bits wide.

The PC stack is popped during returns from interrupts (RTI), returns from subroutines (RTS) and terminations of loops. The stack is full when all entries are occupied, empty when no entries are occupied, and overflowed if a call occurs when the stack is already full. The full and empty flags are stored in the sticky status register (STKY). The full flag causes a maskable interrupt.

A PC stack interrupt occurs when 29 locations of the PC stack are filled (the *almost full* state). Entering the interrupt service routine then immediately causes a push on the PC stack, making it full. Thus the interrupt is a *stack full* interrupt, even though the condition that triggers it is the *almost full* condition. The other stacks in the sequencer, the loop address stack, loop counter stack and status stack, are provided with overflow interrupts that are activated when a push occurs while the stack is in a full state.

The program counter stack pointer (PCSTKP) is a readable and writeable register that contains the address of the top of the PC stack. The value of PCSTKP is zero when the PC stack is empty, 1, 2, ..., 30 when the stack contains data, and 31 when the stack is overflowed. A write to PCSTKP takes effect after a one-cycle delay. If the PC stack is overflowed, a write to PCSTKP has no effect.

3.5 LOOPS (DO UNTIL)

The DO UNTIL instruction provides for efficient software loops, without the overhead of additional instructions to branch, test a condition, or decrement a counter. Here is a simple example of an ADSP-2106x loop:

```
        LCNTR=30, DO label UNTIL LCE;
        R0=DM(I0,M0), F2=PM(I8,M8);
        R1=R0-R15;
label:  F4=F2+F3;
```

When the ADSP-2106x executes a DO UNTIL instruction, the program sequencer pushes the address of the last loop instruction and the termination condition for exiting the loop (both specified in the instruction) onto the loop address stack. It also pushes the top-of-loop address, which is the address of the instruction following the DO UNTIL instruction, on the PC stack.

Because of the instruction pipeline (fetch, decode and execute cycles), the processor tests the termination condition (and, if the loop is counter-based, decrements the counter) before the end of the loop so that the next fetch either exits the loop or returns to the top based on the test condition. Specifically, the condition is tested when the instruction two locations before the last instruction in the loop (at location $e - 2$, where e is the end-of-loop address) is executed. If the termination condition is not satisfied, the processor fetches the instruction from the top-of-loop address stored on the top of the PC stack. If the termination condition is true, the sequencer fetches the next instruction after the end of the loop and pops the loop stack and PC stack. Loop operation is shown in Figure 3.6, on the next page.

LOOP-BACK

CLOCK CYCLES →

Execute Instruction	e-2	e-1	e	b
Decode Instruction	e-1	e	b	b+1
Fetch Instruction	e	b	b+1	b+2
	termination condition tests false	loop start address is top of PC stack		

LOOP TERMINATION

CLOCK CYCLES →

Execute Instruction	e-2	e-1	e	e+1
Decode Instruction	e-1	e	e+1	e+2
Fetch Instruction	e	e+1	e+2	e+3
	termination condition tests true	loop-back aborts; PC and loop stacks popped		

e = Loop end instruction
b = Loop start instruction

Figure 3.6 Loop Operation

3.5.1 Restrictions & Short Loops

This section describes several programming restrictions for loops. It also explains restrictions applying to short (one- and two-instruction) loops, which require special consideration because of the three-instruction fetch-decode-execute pipeline.

3.5.1.1 General Restrictions

- Nested loops cannot terminate on the same instruction.

- The last three instructions of a loop cannot be any branch (jump, call, or return); otherwise, the loop may not be executed correctly. This also applies to one-instruction loops and two-instruction loops with only one iteration. There is one exception to this rule, a non-delayed CALL (no DB modifier) paired with an RTS (LR), return from subroutine with loop reentry modifier. The non-delayed CALL may be used as one of the last three instructions of a loop (but not in a one-instruction loop or a two-instruction, single-iteration loop.)

  The RTS (LR) instruction ensures proper reentry into a loop. In counter-based loops, for example, the termination condition is checked by decrementing the current loop counter (CURLCNTR) during execution of the instruction two locations before the end of the loop. A non-delayed call may then be used in one of the last two locations, providing an RTS (LR) instruction is used to return from the subroutine. The loop reentry (LR) modifier assures proper reentry into the loop, by preventing the loop counter from being decremented again (i.e. twice for the same loop iteration).

3.5.1.2 Counter-Based Loops

The third-to-last instruction of a counter-based loop (at $e - 2$, where e is the end-of-loop address) cannot be a write to the counter from memory.

Short loops terminate in a special way because of the instruction (fetch-decode-execute) pipeline. Counter-based loops of one or two instructions are not long enough for the sequencer to check the termination condition two instructions from the end of the loop. In these short loops, the sequencer has already looped back when the termination condition is tested. The sequencer provides special handling to avoid overhead (NOP) cycles if the loop is iterated a minimum number of times. The detailed operation is shown in Figures 3.7 and 3.8 (on the following page). For no overhead, a loop of length one must be executed at least three times and a loop of length two must be executed at least twice.

Loops of length one that iterate only once or twice and loops of length two that iterate only once incur two cycles of overhead because there are two aborted instructions after the last iteration to clear the instruction pipeline.

Processing of an interrupt that occurs during the last iteration of a one-instruction loop that executes once or twice, a two-instruction loop that executes once, or the cycle following one of these loops (which is a NOP) is delayed by one cycle. Similarly, in a one-instruction loop that iterates at least three times, processing is delayed by one cycle if the interrupt occurs during the third-to-last iteration.

3.5.1.3 Non-Counter-Based Loops

A non-counter-based loop is one in which the loop termination condition is something other than LCE. When a non-counter-based loop is the outer loop of a series of nested loops, the end address of the outer loop must be located at least two addresses after the end address of the inner loop.

The JUMP (LA) instruction is used to prematurely abort execution of a loop. When this instruction is located in the inner loop of a series of nested loops and the outer loop is non-counter-based, the address jumped to cannot be the last instruction of the outer loop. The address jumped to may, however, be the next-to-last instruction (or any earlier).

ONE-INSTRUCTION LOOP, THREE ITERATIONS

CLOCK CYCLES →

Execute Instruction	n	n+1 first iteration	n+1 second iteration	n+1 third iteration	n+2
Decode Instruction	n+1	n+1	n+1	n+2	n+3
Fetch Instruction	n+2	n+1	n+2	n+3	n+4
	LCNTR <– 3	opcode latch not updated; fetch address not updated; count expired tests true	loop-back aborts; PC & loop stacks popped		

ONE-INSTRUCTION LOOP, TWO ITERATIONS (Two Cycles of Overhead)

CLOCK CYCLES →

Execute Instruction	n	n+1 first iteration	n+1 second iteration	nop	nop	n+2
Decode Instruction	n+1	n+1	n+1 -> nop	n+1 –> nop	n+2	n+3
Fetch Instruction	n+2	n+1	n+1	n+2	n+3	n+4
	LCNTR <– 2	opcode latch not updated; fetch address not updated	count expired tests true	loop-back aborts; PC & loop stacks popped		

Figure 3.7 One-Instruction Counter-Based Loops

n = DO UNTIL instruction
n+2 = instruction after loop

Non-counter-based short loops terminate in a special way because of the *fetch-decode-execute* instruction pipeline:

- In a three-instruction loop, the termination condition is tested when the top of loop instruction is executed. When the condition becomes true, the sequencer completes one full pass of the loop before exiting.
- In a two-instruction loop, the termination condition is checked during the last (second) instruction. If the condition becomes true when the first instruction is executed, it tests true during the second and one more full pass is completed before exiting. If the condition becomes true during the second instruction, however, two more full passes occur before the loop exit.
- In a one-instruction loop, the termination condition is checked every cycle. When the condition becomes true, the loop executes three more times before exiting.

TWO-INSTRUCTION LOOP, TWO ITERATIONS

CLOCK CYCLES →

Execute Instruction	n	n+1 first iteration	n+2 first iteration	n+1 second iteration	n+2 second iteration	n+3
Decode Instruction	n+1	n+2	n+1	n+2	n+3	n+4
Fetch Instruction	n+2	n+1	n+2	n+3	n+4	n+5
	LCNTR <- 2	PC stack supplies loop start address	last instruction fetched, causes condition test; tests true	loop-back aborts; PC & loop stacks popped		

TWO-INSTRUCTION LOOP, ONE ITERATION (Two Cycles of Overhead)

CLOCK CYCLES →

Execute Instruction	n	n+1 first iteration	n+2 first iteration	nop	nop	n+3
Decode Instruction	n+1	n+2	n+1->nop	n+2->nop	n+3	n+4
Fetch Instruction	n+2	n+1	n+2	n+3	n+4	n+5
	LCNTR <- 1	PC stack supplies loop start address	last instruction fetched, causes condition test; tests true	loop-back aborts; PC & loop stacks popped		

n = DO UNTIL instruction
n+3 = instruction after loop

Figure 3.8 Two-Instruction Counter-Based Loops

3.5.2 Loop Address Stack

The loop address stack is six levels deep by 32 bits wide. The 32-bit word of each level consists of a 24-bit loop termination address, a 5-bit termination code, and a 2-bit loop type code:

Bits	*Value*
0-23	Loop termination address
24-28	Termination code
29	*reserved (always reads 0)*
30-31	Loop type code:

Code	Loop type
00	arithmetic condition-based (not LCE)
01	counter-based, length 1
10	counter-based, length 2
11	counter-based, length > 2

The loop termination address, termination code and loop type code are stacked when a DO UNTIL or PUSH LOOP instruction is executed. The stack is popped two instructions before the end of the last loop iteration or when a POP LOOP instruction is issued. A stack overflows if a push occurs when all entries in the loop stack are occupied. The stack is empty when no entries are occupied. The overflow and empty flags are in the sticky status register (STKY). Overflow causes a maskable interrupt.

The LADDR register contains the top of the loop address stack. It is readable and writeable over the DM Data bus. Reading and writing LADDR does not move the loop address stack pointer; a stack push or pop, performed with explicit instructions, moves the stack pointer. LADDR contains the value 0xFFFF FFFF when the loop address stack is empty.

Because the termination condition is checked two instructions before the end of the loop, the loop stack is popped before the end of the loop on the final iteration. If LADDR is read at either of these instructions, the value will no longer be the termination address for the loop.

A jump out of a loop pops the loop address stack (and the loop count stack if the loop is counter-based) if the Loop Abort (LA) modifier is specified for the jump. This allows the loop mechanism to continue to function correctly. Only one pop is performed, however, so the loop abort cannot be used to jump more than one level of loop nesting.

3.5.3 Loop Counters And Stack

The loop counter stack is six levels deep by 32 bits wide. The loop counter stack works in synchronization with the loop address stack; both stacks always have the same number of locations occupied. Thus, the same empty and overflow status flags apply to both stacks.

The ADSP-2106x program sequencer operates two separate loop counters: the current loop counter (CURLCNTR), which tracks iterations for a loop being executed, and the loop counter (LCNTR), which holds the count value before the loop is executed. Two counters are needed to maintain the count for an outer loop while setting up the count for an inner loop.

3.5.3.1 CURLCNTR

The top entry in the loop counter stack always contains the loop count currently in effect. This entry is the CURLCNTR register, which is readable and writeable over the DM Data bus. A read of CURLCNTR when the loop counter stack is empty gives the value 0xFFFF FFFF.

The program sequencer decrements the value of CURLCNTR for each loop iteration. Because the termination condition is checked two instruction cycles before the end of the loop, the loop counter is also decremented before the end of the loop. If CURLCNTR is read at either of the last two loop instructions, therefore, the value is already the count for the next iteration.

The loop counter stack is popped two instructions before the end of the last loop iteration. When the loop counter stack is popped, the new top entry of the stack becomes the CURLCNTR value, the count in effect for the executing loop. If there is no executing loop, the value of CURLCNTR is 0xFFFF FFFF after the pop.

Writing CURLCNTR does not cause a stack push. Thus, if you write a new value to CURLCNTR, you change the count value of the loop currently executing. A write to CURLCNTR when no DO UNTIL LCE loop is executing has no effect.

Because the processor must use CURLCNTR to perform counter-based loops, there are some restrictions on when you can write CURLCNTR. As mentioned under “Loop Restrictions,” the third-to-last instruction of a DO UNTIL LCE loop cannot be a write to CURLCNTR from memory. The instruction that follows a write to CURLCNTR from memory cannot be an IF NOT LCE instruction.

3.5.3.2 LCNTR

LCNTR is the value of the top of the loop counter stack *plus one*, i.e., it is the location on the stack which will take effect on the next loop stack push. To set up a count value for a nested loop without affecting the count value of the loop currently executing, you write the count value to LCNTR. A value of zero in LCNTR causes a loop to execute 2^{32} times.

The DO UNTIL LCE instruction pushes the value of LCNTR on the loop count stack, so that it becomes the new CURLCNTR value. This process is illustrated in Figure 3.9. The previous CURLCNTR value is preserved one location down in the stack.

A read of LCNTR when the loop counter stack is full results in invalid data. When the loop counter stack is full, any data written to LCNTR is discarded.

If you read LCNTR during the last two instructions of a terminating loop, its value is the last CURLCNTR value for the loop.

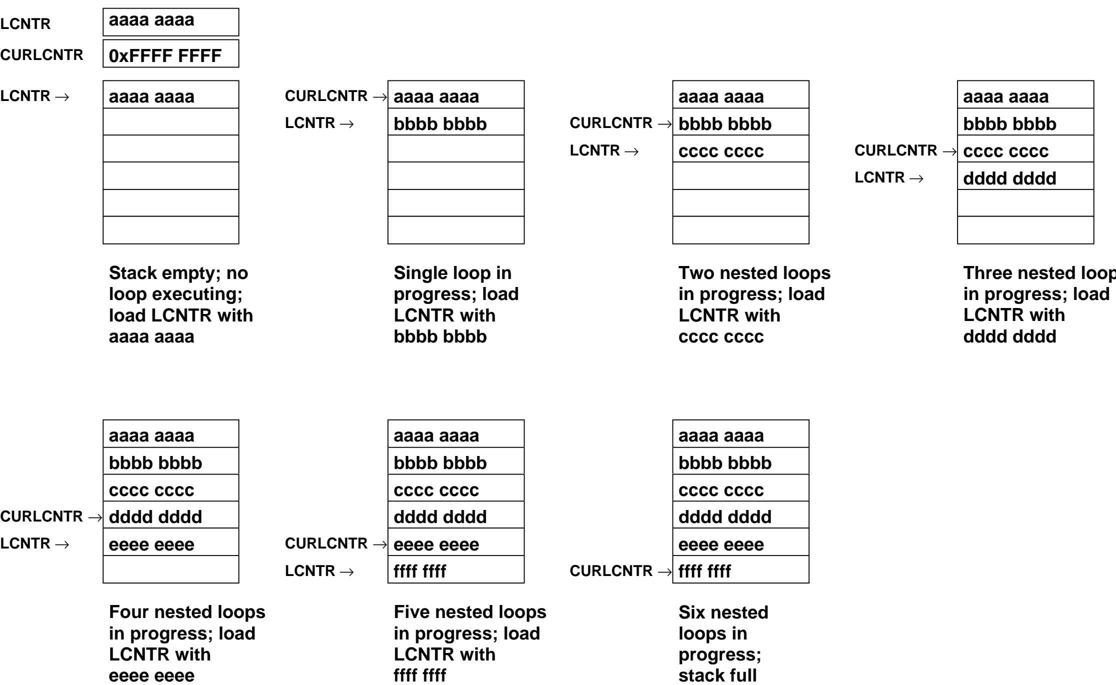


Figure 3.9 Pushing The Loop Counter Stack For Nested Loops

3.6 INTERRUPTS

Interrupts are caused by a variety of conditions, both internal and external to the processor. An interrupt forces a subroutine call to a predefined address, the interrupt vector. The ADSP-2106x assigns a unique vector to each type of interrupt.

Externally, the ADSP-2106x supports three prioritized, individually maskable interrupts, each of which can be either level or edge-triggered. These interrupts are caused by an external device asserting one of the ADSP-2106x's interrupt inputs ($\overline{IRQ}_{2-0}$). Among the internally generated interrupts are arithmetic exceptions, stack overflows, and circular data buffer overflows.

An interrupt request is deemed valid if it is not masked, if interrupts are globally enabled (if bit 12 in MODE1 is set), and if a higher priority request is not pending. Valid requests invoke an interrupt service sequence that branches to the address reserved for that interrupt. Interrupt vectors are spaced at 8-instruction intervals; longer service routines can be accommodated by branching to another region of memory. Program execution returns to normal sequencing when an RTI (return from interrupt) instruction is executed.

The ADSP-2106x core processor cannot service an interrupt unless it is executing instructions or is in the IDLE state. IDLE and IDLE16 are a special instructions that halt the processor core until an external interrupt or the timer interrupt occurs.

To process an interrupt, the ADSP-2106x's program sequencer performs the following actions:

1. Outputs the appropriate interrupt vector address.

2. Pushes the current PC value (the return address) on the PC stack.

3. If the interrupt is either an external interrupt ($\overline{IRQ}_{2-0}$), the internal timer interrupt, or the VIRPT multiprocessor vector interrupt, the program sequencer pushes the current value of the ASTAT and MODE1 registers onto the status stack.

4. Sets the appropriate bit in the interrupt latch register (IRPTL).

5. Alters the interrupt mask pointer (IMASKP) to reflect the current interrupt nesting state. The nesting mode (NESTM) bit in the MODE1 register determines whether all interrupts or only lower priority interrupts are masked during the service routine.

At the end of the interrupt service routine, the RTI instruction causes the following actions:

1. Returns to the address stored at the top of the PC stack.

2. Pops this value off of the PC stack.

3. Pops the status stack if the ASTAT and MODE1 status registers were pushed (for the $\overline{IRQ}_{2-0}$ external interrupts, timer interrupt, or VIRPT vector interrupt).

4. Clears the appropriate bit in the interrupt latch register (IRPTL) and interrupt mask pointer (IMASKP).

All interrupt service routines, except for reset, should end with a return-from-interrupt (RTI) instruction. After reset, the PC stack is empty, so there is no return address—the last instruction of the reset service routine should be a jump to the start of your program.

3.6.1 Interrupt Latency

The ADSP-2106x responds to interrupts in three stages: synchronization and latching (1 cycle), recognition (1 cycle), and branching to the interrupt vector (2 cycles). See Figure 3.10. If an interrupt is forced in software by a write to a bit in IRPTL, it is recognized in the following cycle, and the two cycles of branching to the interrupt vector follow that.

For most interrupts, internal and external, only one instruction is executed after the interrupt occurs (and before the two instructions aborted) while the processor fetches and decodes the first instruction of the service routine. Because of the one-cycle delay between an arithmetic exception and the STKY register update, however, there are two cycles after an arithmetic exception occurs before interrupt processing starts.

The standard latency associated with the $\overline{IRQ}_{2-0}$ interrupts and the multiprocessor vector interrupt are:

Interrupt	*Latency (minimum)*
$\overline{IRQ}_{2-0}$ interrupts	3 cycles
Multiprocessor vector interrupt (VIRPT register)	6 cycles

INTERRUPT, SINGLE-CYCLE INSTRUCTION — n = Single-cycle instruction

CLOCK CYCLES ⟶

Execute Instruction	n-1	n	nop	nop	v
Decode Instruction	n	n+1->nop	n+2->nop	v	v+1
Fetch Instruction	n+1	n+2	v	v+1	v+2
	interrupt occurs	interrupt recognized	n+1 pushed onto PC stack; interrupt vector output		

INTERRUPT, PROGRAM MEMORY DATA ACCESS WITH CACHE MISS — n = Instruction coinciding with program memory data access, cache miss

CLOCK CYCLES ⟶

Execute Instruction	n-1	n	nop	nop	nop	v
Decode Instruction	n	n+1->nop	n+1->nop	n+2->nop	v	v+1
Fetch Instruction	n+1	-	n+2	v	v+1	v+2
	interrupt occurs	interrupt recognized, but not processed; program memory data access	interrupt processed	n+1 pushed onto PC stack; interrupt vector output		

INTERRUPT, DELAYED BRANCH — n = Delayed branch instruction

CLOCK CYCLES ⟶

Execute Instruction	n-1	n	n+1	n+2	nop	nop	v
Decode Instruction	n	n+1	n+2	j->nop	j+1 ->nop	v	v+1
Fetch Instruction	n+1	n+2	j	j + 1	v	v+1	v+2
	interrupt occurs	interrupt recognized, but not processed		for a call, n+3 pushed onto PC stack; interrupt processed	j pushed onto PC stack; interrupt vector output		

Figure 3.10 Interrupt Handling

v = instruction at interrupt vector
j = instruction at branch address

If nesting is enabled and a higher priority interrupt occurs immediately after a lower priority interrupt, the service routine of the higher priority interrupt is delayed by one additional cycle. (See "Interrupt Nesting & IMASKP".) This allows the first instruction of the lower priority interrupt routine to be executed before it is interrupted.

Certain ADSP-2106x operations that span more than one cycle will hold off interrupt processing. If an interrupt occurs during one of these operations, it is synchronized and latched, but its processing is delayed. The operations that delay interrupt processing in this way are as follows:

- a branch (call, jump, or return) and the following cycle, whether it is an instruction (in a delayed branch) or a NOP (in a non-delayed branch)
- the first of the two cycles needed to perform a program memory data access and an instruction fetch (when there is an instruction cache miss).
- the third-to-last iteration of a one-instruction loop
- the last iteration of a one-instruction loop executed once or twice or of a two-instruction loop executed once, and the following cycle (which is a NOP)
- the first of the two cycles needed to fetch and decode the first instruction of an interrupt service routine
- waitstates for external memory accesses
- when an external memory access is required and the ADSP-2106x does not have control of the external bus (during a host bus grant or when the ADSP-2106x is a bus slave in a multiprocessing system)

3.6.2 Interrupt Vector Table

Table 3.3 shows all ADSP-2106x interrupts, listed according their bit position in the IRPTL and IMASK registers (see "Interrupt Latch Register"). Also shown is the address of the interrupt vector; each vector is separated by eight memory locations. The addresses in the vector table represent offsets from a base address. For an interrupt vector table in internal memory, the base address is 0x0002 0000; for an interrupt vector table in external memory, the base address is 0x0040 0000. The third column in Table 3.3 lists a mnemonic name for each interrupt. These names are provided for convenience, and are not required by the assembler.

IRPTL/ IMASK Bit #	*Vector Address**	*Interrupt Name***	*Function*	
0	0x00	–	*reserved*	
1	0x04	RSTI	Reset (read-only, non-maskable)	**HIGHEST PRIORITY**
2	0x08	–	*reserved*	
3	0x0C	SOVFI	Status stack or loop stack overflow or PC stack full	
4	0x10	TMZHI	Timer=0 (high priority option)	
5	0x14	VIRPTI	Vector Interrupt	
6	0x18	IRQ2I	$\overline{\text{IRQ2}}$ asserted	
7	0x1C	IRQ1I	$\overline{\text{IRQ1}}$ asserted	
8	0x20	IRQ0I	$\overline{\text{IRQ0}}$ asserted	
9	0x24	–	*reserved*	
10	0x28	SPR0I	DMA Channel 0 – SPORT0 Receive	
11	0x2C	SPR1I	DMA Channel 1 – SPORT1 Receive (or Link Buffer 0)	
12	0x30	SPT0I	DMA Channel 2 – SPORT0 Transmit	
13	0x34	SPT1I	DMA Channel 3 – SPORT1 Transmit (or Link Buffer 1)	
14	0x38	LP2I	DMA Channel 4 – Link Buffer 2	
15	0x3C	LP3I	DMA Channel 5 – Link Buffer 3	
16	0x40	EP0I	DMA Channel 6 – Ext. Port Buffer 0 (or Link Buffer 4)	
17	0x44	EP1I	DMA Channel 7 – Ext. Port Buffer 1 (or Link Buffer 5)	
18	0x48	EP2I	DMA Channel 8 – Ext. Port Buffer 2	
19	0x4C	EP3I	DMA Channel 9 – Ext. Port Buffer 3	
20	0x50	LSRQ	Link Port Service Request	
21	0x54	CB7I	Circular Buffer 7 overflow	
22	0x58	CB15I	Circular Buffer 15 overflow	
23	0x5C	TMZLI	Timer=0 (low priority option)	
24	0x60	FIXI	Fixed-point overflow	
25	0x64	FLTOI	Floating-point overflow exception	
26	0x68	FLTUI	Floating-point underflow exception	
27	0x6C	FLTII	Floating-point invalid exception	
28	0x70	SFT0I	User software interrupt 0	
29	0x74	SFT1I	User software interrupt 1	
30	0x78	SFT2I	User software interrupt 2	
31	0x7C	SFT3I	User software interrupt 3	**LOWEST PRIORITY**

Table 3.3 Interrupt Vectors & Priority

* Offset from base address: 0x0002 0000 for interrupt vector table in internal memory, 0x0040 0000 for interrupt vector table in external memory

** These IRPTL/IMASK bit names are defined in the `def21060.h` include file supplied with the ADSP-21000 Family Development Software.

The interrupt vector table may be located in internal memory, at address 0x0002 0000 (the beginning of Block 0), or in external memory at address 0x0040 0000. If the ADSP-2106x's on-chip memory is booted from an external source, the interrupt vector table will be located in internal memory. If, however, the ADSP-2106x is not booted (because it will execute from off-chip memory), the vector table must be located in the off-chip memory. See "Booting" in the *System Design* chapter for details on booting mode selection.

Also, if booting is from an external EPROM or host processor, bit 16 of IMASK (the EP0I interrupt for external port DMA Channel 6) will automatically be set to 1 following reset—this enables the *DMA done* interrupt for booting on Channel 6. IRPTL is initialized to all zeros following reset.

The IIVT bit in the SYSCON control register can be used to override the booting mode in determining where the interrupt vector table is located. If the ADSP-2106x is not booted (*no boot* mode), setting IIVT to 1 selects an internal vector table while IIVT=0 selects an external vector table. If the ADSP-2106x is booted from an external source (any mode other than *no boot* mode), then IIVT has no effect.

3.6.3 Interrupt Latch Register (IRPTL)

The interrupt latch (IRPTL) register is a 32-bit register that latches interrupts. It indicates all interrupts currently being serviced as well as any which are pending. Because this register is readable and writeable, any interrupt (except reset) can be set or cleared in software. Do not write to the reset bit (bit 1) in IRPTL because this puts the processor into an illegal state.

When an interrupt occurs, the corresponding bit in IRPTL is set. During execution of the interrupt's service routine, this bit is kept cleared—the ADSP-2106x clears the bit during every cycle, preventing the same interrupt from being latched while its service routine is already executing.

A special method is provided, however, to allow the reuse of an interrupt while it is being serviced. This method is provided by the clear interrupt (CI) modifier of the JUMP instruction. See Section 3.6.8, "Clearing The Current Interrupt For Reuse."

IRPTL is cleared by a processor reset.

(Note: The bits in the IMASK register correspond exactly to those in IRPTL.)

3.6.4 Interrupt Priority

The interrupt bits in IRPTL are ordered by priority. The interrupt priority is from 0 (highest) to 31 (lowest). Interrupt priority determines which interrupt is serviced first when more than one occurs in the same cycle. It also determines which interrupts are nested when nesting is enabled (see "Interrupt Nesting and IMASKP").

The arithmetic interrupts—fixed-point overflow and floating-point overflow, underflow, and invalid operation—are determined from flags in the sticky status register (STKY). By reading these flags, the service routine for one of these interrupts can determine which condition caused the interrupt. The routine also has to clear the appropriate STKY bit so that the interrupt is not still active after the service routine is done.

The timer decrementing to zero causes both interrupt 4 and interrupt 14. This feature allows you to choose the priority of the timer interrupt. Unmask the timer interrupt that has the priority you want, and leave the other one masked. Unmasking both interrupts results in two interrupts when the timer reaches zero. In this case the processor services the higher priority interrupt first, then the lower priority interrupt.

3.6.5 Interrupt Masking & Control

All interrupts except for reset can be enabled and disabled by the global interrupt enable bit, IRPTEN, bit 12 in the MODE1 register. This bit is cleared at reset. You must set this bit for interrupts to be enabled.

3.6.5.1 Interrupt Mask Register (IMASK)

All interrupts except for reset can be masked. Masked means the interrupt is disabled. Interrupts that are masked are still latched (in IRPTL), so that if the interrupt is later unmasked, it is processed.

The IMASK register controls interrupt masking. The bits in IMASK correspond exactly to the bits in the IRPTL register. For example, bit 10 in IMASK masks or unmasks the same interrupt latched by bit 10 in IRPTL.

– *If a bit in IMASK is set to 1, its interrupt is unmasked (enabled).*
– *If the bit is cleared (to 0), the interrupt is masked (disabled).*

After reset, all interrupts except for the reset interrupt and the EP0I interrupt for external port DMA Channel 6 (bit 16 of IMASK) are masked. The reset interrupt is always non-maskable. The EP0I interrupt is automatically unmasked after reset if the ADSP-2106x is booting from EPROM or from a host.

3.6.5.2 Interrupt Nesting & IMASKP

The ADSP-2106x supports the nesting of one interrupt service routine inside another; that is, a service routine can be interrupted by a higher priority interrupt. This feature is controlled by the nesting mode bit (NESTM) in the MODE1 register.

When the NESTM bit is a 0, an interrupt service routine cannot be interrupted; any interrupt that occurs will be processed only after the routine finishes. When NESTM is a 1, higher priority interrupts can interrupt if they are not masked; lower or equal priority interrupts cannot. The NESTM bit should only be changed outside of an interrupt service routine or during the reset service routine; otherwise, interrupt nesting may not work correctly.

If nesting is enabled and a higher priority interrupt occurs immediately after a lower priority interrupt, the service routine of the higher priority interrupt is delayed by one cycle. This allows the first instruction of the lower priority interrupt routine to be executed before it is interrupted.

In nesting mode, the ADSP-2106x uses the interrupt mask pointer (IMASKP) to create a temporary interrupt mask for each level of interrupt nesting; the IMASK value is not affected. The ADSP-2106x changes IMASKP each time a higher priority interrupt interrupts a lower priority service routine.

The bits in IMASKP correspond to the interrupts in order of priority, the same as in IRPTL and IMASK. When an interrupt occurs, its bit is set in IMASKP. If nesting is enabled, a new temporary interrupt mask is generated by masking all interrupts of equal or lower priority to the highest priority bit set in IMASKP (and keeping higher priority interrupts the same as in IMASK). When a return from an interrupt service routine (RTI) is executed, the highest priority bit set in IMASKP is cleared, and again a new temporary interrupt mask is generated by masking all interrupts of equal or lower priority to the highest priority bit set in IMASKP. The bit set in IMASKP that has the highest priority always corresponds to the priority of the interrupt being serviced.

If nesting is not enabled, the processor masks out all interrupts and IMASKP is not used, although IMASKP is still updated to create a temporary interrupt mask.

IRPTL is updated, but the ADSP-2106x does not vector to an interrupt that occurs while its service routine is already executing. It waits until the RTI completes before vectoring to the service routine again.

3.6.6 Status Stack Save & Restore

For low-overhead interrupt servicing, the ADSP-2106x automatically saves and restores the status and mode contexts of the interrupted program. The three external interrupts ($\overline{IRQ}_{2-0}$), the timer interrupt, and the VIRPT vector interrupt cause an automatic push of ASTAT and MODE1 onto the status stack, which is five levels deep. These registers are automatically popped from the status stack by the return from interrupt instruction, RTI (and by the JUMP (CI) instruction, described below in "Clearing The Current Interrupt For Reuse").

➠ Only $\overline{IRQ}_{2-0}$, timer, and VIRPT interrupts cause a push of the status stack. All other interrupts require an explicit save and restore of the appropriate registers to memory.

Pushing ASTAT and MODE1 preserves the status and control bit settings so that if the service routine alters these bits, the original settings are automatically restored upon the return from interrupt.

Note, however, that the $FLAG_{3-0}$ bits in ASTAT are not affected by status stack pushes and pops; the values of these bits carry over from the main program to the service routine and from the service routine back to the main program.

The top of the status stack contains the current values of ASTAT and MODE1. Reading and writing these registers does not move the stack pointer. The stack pointer is moved, however, by explicit PUSH and POP instructions.

3.6.7 Software Interrupts

The ADSP-2106x provides software interrupts that emulate interrupt behavior but are activated through software instead of hardware. Setting one of bits 28-31 in IRPTL, with either a BIT SET instruction or a write to IRPTL, activates a software interrupt. The ADSP-2106x branches to the corresponding interrupt routine if that interrupt is not masked and interrupts are enabled.

3.6.8 Clearing The Current Interrupt For Reuse

Normally the ADSP-2106x ignores and does not latch an interrupt that reoccurs while its service routine is already executing. When the interrupt initially occurs, the corresponding bit in IRPTL is set. During execution of the service routine, this bit is kept cleared—the ADSP-2106x clears the bit during every cycle, preventing the same interrupt from being latched while its service routine is already executing.

The clear interrupt (CI) modifier of the JUMP instruction, however, allows the reuse of an interrupt while it is being serviced. This can be useful in systems that require fast interrupt response and low interrupt latency. The JUMP (CI) instruction should be located within the interrupt service routine. JUMP (CI) clears the status of the current interrupt without leaving the interrupt service routine, reducing the interrupt routine to a normal subroutine—this allows the interrupt to occur again, as a result of a different event or task in the ADSP-2106x system.

The JUMP (CI) instruction reduces an interrupt service routine to a normal subroutine by clearing the appropriate bit in the interrupt latch register (IRPTL) and interrupt mask pointer (IMASKP) and popping the status stack. The ADSP-2106x then stops automatically clearing the interrupt's latch bit (in IRPTL) in every cycle, allowing the interrupt to occur again.

When returning from a subroutine which has been reduced from an interrupt service routine with a JUMP (CI) instruction, the (LR) modifier of the RTS instruction must be used (in case the interrupt occurred during the last two instructions of a loop). Refer to "General Restrictions" in Section 3.5, "Loops", for a description of the RTS (LR) instruction.

The following example shows an interrupt service routine that is reduced to a subroutine with the (CI) modifier:

```
instr1;                  {interrupt entry from main program}
JUMP(PC,3) (DB,CI);      {clear interrupt status}
instr3;
instr4;
instr5;
RTS (LR);        {use LR modifier with return from subroutine}
```

Note that the `JUMP(PC,3)(DB,CI)` instruction actually only continues linear execution flow by jumping to the location PC + 3 (instr5), with the two intervening instructions (instr3, instr4) being executed because of the delayed branch (DB). This JUMP instruction is only an example—a JUMP (CI) can be to any location.

3.6.9 External Interrupt Timing & Sensitivity

Each of the ADSP-2106x's three external interrupts, $\overline{IRQ}_{2\text{-}0}$, can be either level- or edge-triggered.

The ADSP-2106x samples interrupts once every CLKIN cycle. Level-sensitive interrupts are considered valid if sampled active (low). A level-sensitive interrupt must go inactive (high) before the processor returns from the interrupt service routine. If a level-sensitive interrupt is still active when the processor samples it, the processor treats it as a new request, repeating the same interrupt routine without returning to the main program (assuming no higher priority interrupts are active).

Edge-triggered interrupt requests are considered valid if sampled high in one cycle and low in the next. The interrupt can stay active indefinitely. To request another interrupt, the signal must go high, then low again.

Edge-triggered interrupts require less external hardware compared to level-sensitive requests since there is never a need to negate the request. However, multiple interrupting devices may share a single level-sensitive request line on a wired-OR basis, which allows for easy system expansion.

A bit for each interrupt in the MODE2 register indicates the sensitivity mode of each interrupt.

MODE2

Bit	*Name*	*Definition*
0	IRQ0E	1=edge-sensitive; 0=level-sensitive
1	IRQ1E	1=edge-sensitive; 0=level-sensitive
2	IRQ2E	1=edge-sensitive; 0=level-sensitive

3.6.9.1 *Asynchronous External Interrupts*

The processor accepts interrupts that are asynchronous to the ADSP-2106x clock; that is, an interrupt signal may change at any time. An asynchronous interrupt must be held low at least one CLKIN cycle to guarantee that it is sampled. Synchronous interrupts need only meet the setup and hold time requirements relative to the rising edge of CLKIN, as specified in the *ADSP-2106x Data Sheet*.

3.6.10 Multiprocessor Vector Interrupts (VIRPT)

Vector interrupts are used for interprocessor commands in multiple-processor systems. When an external processor writes an address to the VIRPT register a vector interrupt is caused. The external processor may be either another ADSP-2106x or a host.

When the vector interrupt is serviced, the ADSP-2106x automatically pushes the status stack and begins executing the service routine located at the address specified in VIRPT. The lower 24 bits of VIRPT contain the address; the upper 8 bits may be optionally used as data to be read by the interrupt service routine. At reset, VIRPT is initialized to its standard address in the ADSP-2106x's interrupt vector table.

The minimum latency for vector interrupts is six cycles, five of which are NOPs. When the RTI (return from interrupt) instruction is reached in the service routine, the ADSP-2106x automatically pops the status stack.

The VIPD bit in SYSTAT reflects the status of the VIRPT register. If VIRPT is written while a previous vector interrupt is pending, the new vector address replaces the pending one. If VIRPT is written while a previous vector interrupt is being serviced, the new vector address is ignored and no new interrupt is triggered. If the ADSP-2106x writes to its own VIRPT register it is ignored.

To use the ADSP-2106x's vector interrupt feature, external processors can take the following sequence of actions:

1. Poll the VIRPT register until it reads a certain token value (i.e. zero).
2. Write the vector interrupt service routine address to VIRPT.
3. When the service routine is finished, it writes the token back into VIRPT to indicate that it is finished and that another vector interrupt can be initiated.

3.7 TIMER

The ADSP-2106x includes a programmable interval timer which can generate periodic interrupts. You program the timer by writing values to its two registers and you control timer operation through a bit in the MODE2 register. An external output, TIMEXP, signals to other devices that the timer count has expired.

Figure 3.11 shows a block diagram of the timer. Two universal registers, TPERIOD and TCOUNT, control the timer interval.

Register	*Function*	*Width*
TPERIOD	Timer Period Register	32 bits
TCOUNT	Timer Counter Register	32 bits

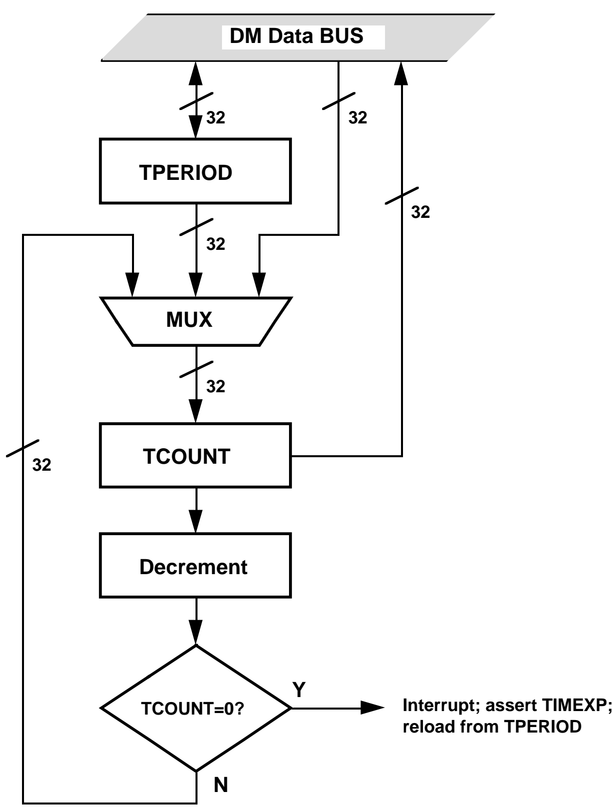


Figure 3.11 Timer Block Diagram

The TCOUNT register contains the timer counter. The timer decrements the TCOUNT register each clock cycle. When the TCOUNT value reaches zero, the timer generates an interrupt and asserts the TIMEXP output high for 4 cycles (when the timer is enabled). See Figure 3.12. On the clock cycle after TCOUNT reaches zero, the timer automatically reloads TCOUNT from the TPERIOD register.

The TPERIOD value specifies the frequency of timer interrupts. The number of cycles between interrupts is TPERIOD + 1. The maximum value of TPERIOD is $2^{32} - 1$, so if the clock cycle is 50 ns, the maximum interval between interrupts is 214.75 seconds.

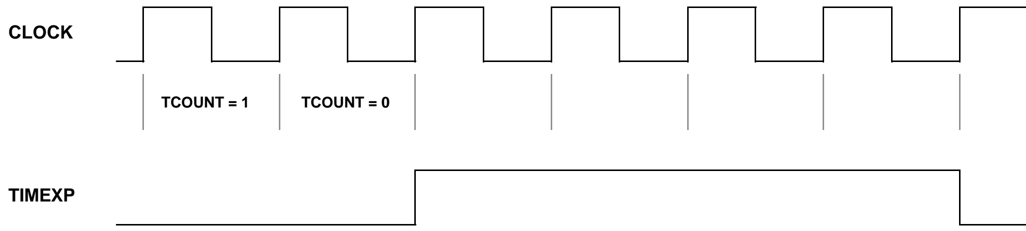


Figure 3.12 TIMEXP Signal

3.7.1 Timer Enable/Disable

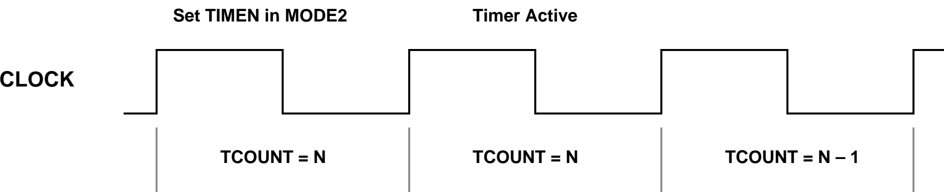
To start and stop the timer, you enable and disable it with the TIMEN bit in the MODE2 register. With the timer disabled, you load TCOUNT with an initial count value and TPERIOD with the number of cycles for the interval you want. Then you enable the timer when you want to begin the count.

At reset, the timer enable bit in the MODE2 register is cleared, so the timer is disabled. When the timer is disabled, it does not decrement the TCOUNT register and it generates no interrupts. When the timer enable bit is set, the timer starts decrementing the TCOUNT register at the end of the next clock cycle. If the bit is subsequently cleared, the timer is disabled and stops decrementing TCOUNT after the next clock cycle (see Figure 3.13).

MODE2

Bit	*Name*	*Definition*
5	TIMEN	Timer enable

TIMER ENABLE



TIMER DISABLE

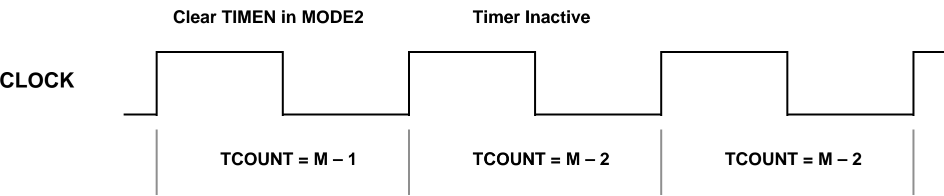


Figure 3.13 Timer Enable & Disable

3.7.2 Timer Interrupts

When the value of TCOUNT reaches zero, the timer generates two interrupts, one with a relatively high priority, the other with a relatively low priority. At reset, both are masked. You should unmask only the timer interrupt that has the priority you want, and leave the other masked.

IRPTL Bit	*Interrupt Name*	*Vector Address*	*Function*
4	TMZHI	0x10	Timer =0 (high priority option)
23	TMZLI	0x5C	Timer=0 (low priority option)

Interrupt priority determines which interrupt is serviced first when two occur in the same cycle. It also affects interrupt nesting—when nesting is enabled, only higher priority interrupts can interrupt a service routine in progress.

Like other interrupts, the timer interrupt requires two cycles to fetch and decode the first instruction of the service routine. The service routine begins executing four cycles after the timer count reaches zero, as shown in Figure 3.14.

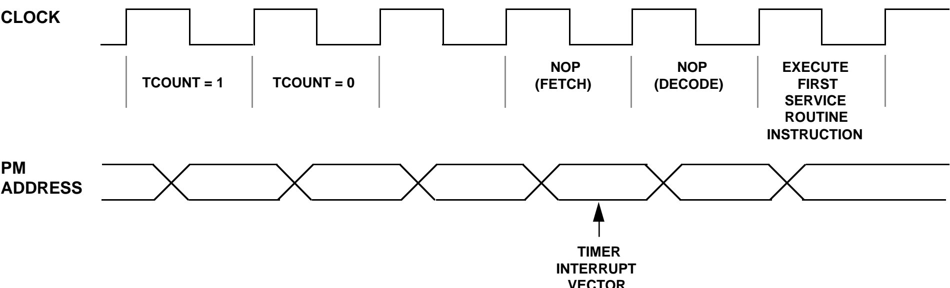


Figure 3.14 Timer Interrupt Timing

3.7.3 Timer Registers

Both the TPERIOD and TCOUNT registers can be read and written by universal register transfers. Reading the registers has no effect on the timer. An explicit write to TCOUNT has priority over both the loading of TCOUNT from TPERIOD and the decrementing of TCOUNT.

Neither TCOUNT nor TPERIOD are affected by a reset, so you should initialize both registers after reset, before enabling the timer.

3.8 STACK FLAGS

The STKY status register maintains stack full and stack empty flags for the PC stack as well as overflow and empty flags for the status stack and loop stack. Unlike other bits in STKY, several of these flag bits are not "sticky." They are set by the occurrence of the corresponding condition and are cleared when the condition is changed (by a push, pop, or processor reset).

STKY Bit	*Name*	*Definition*	*Sticky/ Not Sticky*	*Cleared By*
21	PCFL	PC stack full	Not sticky	Pop
22	PCEM	PC stack empty	Not sticky	Push
23	SSOV	Status stack overflow	Sticky	RESET
24	SSEM	Status stack empty	Not sticky	Push
25	LSOV	Loop stacks* overflow	Sticky	RESET
26	LSEM	Loop stacks* empty	Not sticky	Push

* Loop address stack and loop counter stack

The status stack flags are read-only. Writes to the STKY register have no effect on these bits.

The overflow and full flags are provided for diagnostic aid only and are not intended to allow recovery from overflow. Status stack or loop stack overflow or PC stack full causes an interrupt.

The empty flags facilitate stack saves to memory. You monitor the empty flag when saving a stack to memory to know when all values have been transferred. The empty flags do not cause interrupts because an empty stack is an acceptable condition.

3.9 IDLE & IDLE16

IDLE and IDLE16 are special instructions that halt the ADSP-2106x core processor in a low-power state until an external interrupt ($\overline{IRQ}_{2-0}$), timer interrupt, DMA interrupt, or VIRPT vector interrupt occurs. When the processor executes an IDLE instruction, it fetches one more instruction at the current fetch address and then suspends operation. The ADSP-2106x's I/O processor is unaffected by the IDLE instruction—any DMA transfers to or from internal memory will continue uninterrupted.

The processor's internal clock continues to run during IDLE, as well as the timer (if it is enabled). When an external interrupt ($\overline{IRQ}_{2-0}$), timer interrupt, DMA interrupt, or VIRPT vector interrupt occurs, the processor responds normally. After two cycles needed to fetch and decode the first instruction of the interrupt service routine, the processor continues executing instructions normally.

On the ADSP-21061 only, the IDLE16 instruction executes a NOP and puts the processor in a low power state. IDLE16 is a lower power version of the IDLE instruction. This instruction halts the processor like the IDLE instruction; in this case, the internal clock runs at 1/16th the rate of CLKIN. The ADSP-21061's I/O processor continues to function, but all operations occur at 1/16th the rate. All internal memory transfers require an extra 15 cycles. The serial clocks and frame syncs (if being sourced by the ADSP-21061) are divided down by a factor of 16 during IDLE16. Similarly, all Host accesses take 16 times longer to complete. The processor remains in the low power state until an interrupt occurs.

After returning from the interrupt, execution continues at the instruction following the IDLE or IDLE16 instruction.

3.10 INSTRUCTION CACHE

The ADSP-2106x's on-chip instruction cache is a 2-way, set-associative cache with entries for 32 instructions. Operation of the cache is transparent to the programmer. The ADSP-2106x caches only instructions that conflict with program memory data accesses (over the PM Data Bus, with the address generated by DAG2 on the PM Address Bus). This feature makes the cache considerably more efficient than a cache that loads every instruction, since typically only a few instructions must access data from a block of program memory.

Because of the three-stage instruction pipeline, if the instruction at address *n* requires a program memory data access, there is a conflict with the instruction fetch at address *n*+2, assuming sequential execution. It is this fetched instruction (*n*+2) that is stored in the instruction cache, not the instruction requiring the program memory data access.

If the instruction needed is in the cache, a "cache hit" occurs—the cache provides the instruction while the program memory data access is performed. If the instruction needed is not in the cache, a "cache miss" occurs, and the instruction fetch (from memory) takes place in the cycle following the program memory data access, incurring one cycle of overhead. This instruction is loaded into the cache, if the cache is enabled and not frozen, so that it is available the next time the same instruction (requiring program memory data) is executed.

3.10.1 Cache Architecture

Figure 3.15 shows a block diagram of the instruction cache. The cache contains 32 entries. An entry consists of a register pair containing an instruction and its address. Each entry has a "valid" bit which is set if the entry contains a valid instruction.

The entries are divided into 16 sets (numbered 15-0) of two entries each, entry 0 and entry 1. Each set has an LRU (Least Recently Used) bit whose value indicates which of the two entries contains the least recently used instruction (1=entry 1, 0=entry 0).

Every possible instruction address is mapped to a set in the cache by its 4 LSBs. When the processor needs to fetch an instruction from the cache, it uses the 4 address LSBs as an index to a particular set. Within

LRU Bit Instruction Address Valid Bit

Set 0
Set 1
Set 2
Set 13
Set 14
Set 15

Figure 3.15 Instruction Cache Architecture

that set, it checks the addresses of the two entries to see whether either contains the needed instruction. A cache hit occurs if the instruction is found, and the LRU bit is updated if necessary to indicate the entry that did not contain the needed instruction.

A cache miss occurs if neither entry in the set contains the needed instruction. In this case, a new instruction and its address are loaded into the least recently used entry of the set that matches the 4 LSBs of the address. The LRU bit is toggled to indicate that the other entry in the set is now the least recently used.

Because instructions are mapped to sets by their 4 address LSBs, there is no need to store these bits in the cache; the 4 LSBs are implied by the set in which the instruction has been stored. Only bits 23-4 are actually stored in a cache entry.

3.10.2 Cache Efficiency

Usually, cache operation and its efficiency is not a concern. However, there are some situations that can degrade cache efficiency and can be remedied easily in your program.

When a cache miss occurs, the needed instruction is loaded into the cache so that if the same instruction is needed again, it will be there (i.e. a cache hit will occur). However, if another instruction whose address is mapped to the same set displaces this instruction, there will be a cache miss instead. The LRU bits help to reduce this possibility since at least two other instructions mapped to the same set must be needed before an instruction is displaced. If three instructions mapped to the same set are all needed repeatedly, cache efficiency (i.e. "hit rate") can go to zero. The solution is to move one or more of the instructions to a new address, one that is mapped to a different set.

An example of cache-inefficient code is shown in Figure 3.16. The program memory data access at address 0x101 in the *tight* loop causes the instruction at 0x103 to be cached (in set 3). Each time the subroutine *sub* is called, the program memory data accesses at 0x201 and 0x211 displace this instruction by loading the instructions at 0x203 and 0x213 into set 3. If the subroutine is called only rarely during the loop execution, the impact will be minimal. If the subroutine is called frequently, the effect will be noticeable. If the execution of the loop is time-critical, it would be advisable to move the subroutine up one location (starting at 0x201), so that the two cached instructions end up in set 4 instead of 3.

```
Address
0x0100         lcntr=1024, do tight until lce;
0x0101            r0=dm(i0,m0), pm(i8,m8)=f3;
0x0102            r1=r0-r15;
0x0103            if eq call (sub);
0x0104            f2=float r1;
0x0105            f3=f2*f2;
0x0106  tight: f3=f3+f4;
0x0107         pm(i8,m8)=f3;
•
•
•
0x0200  sub:   r1=R13;
0x0201         r14=pm(i9,m9);
•
•
•
0x0211         pm(i9,m9)=r12;
•
•
•
0x021F         rts;
```


Figure 3.16 Cache-Inefficient Code

3.10.3 Cache Disable & Cache Freeze

Freezing the cache prevents any changes to its contents—a cache miss will not result in a new instruction being stored in the cache. Disabling the cache stops its operation completely; all instruction fetches conflicting with program memory data accesses are delayed by the access. These functions are selected by the CADIS (cache enable/disable) and CAFRZ (cache freeze) bits in the MODE2 register:

MODE2

Bit	*Name*	*Function*
4	CADIS	Cache Disable
19	CAFRZ	Cache Freeze

After reset the cache is cleared, containing no instructions, and is unfrozen and enabled.

An instruction containing a program memory data access must not be placed directly after a cache enable or cache disable instruction—the ADSP-2106x must wait at least one cycle before executing the PM data access. A NOP may be inserted to accomplish this.