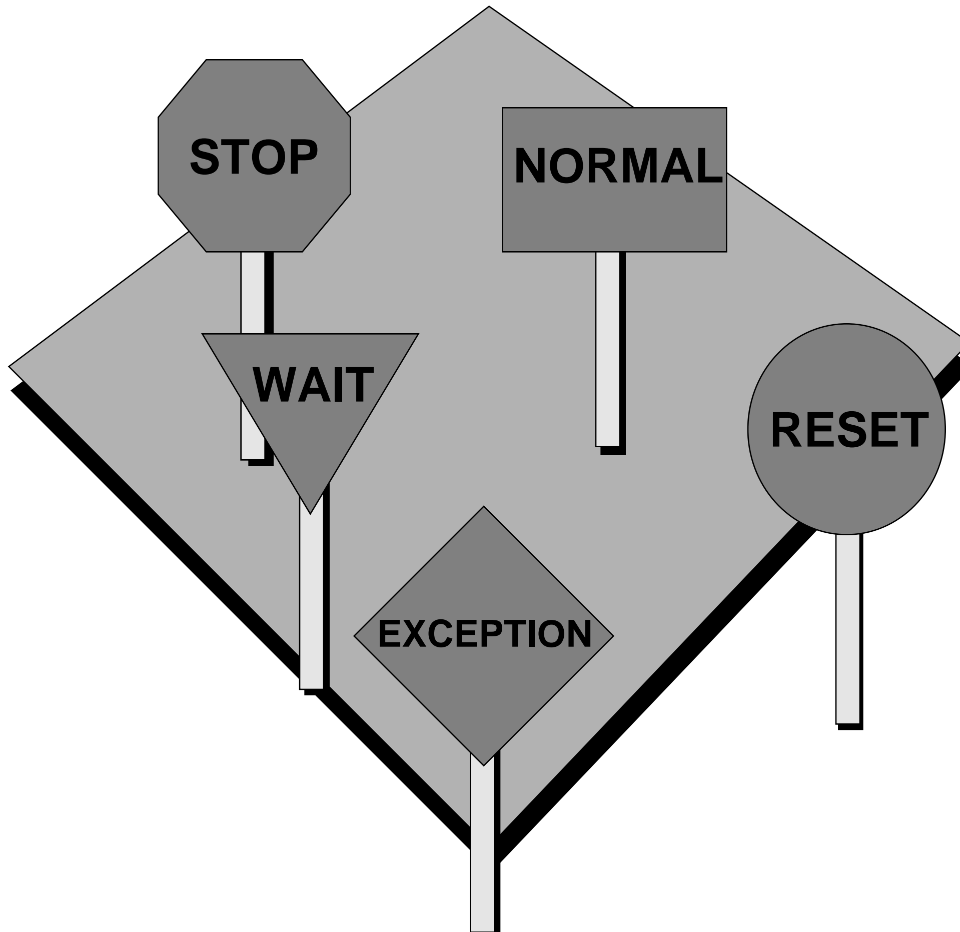
SECTION 7
PROCESSING STATES



SECTION CONTENTS

SECTION 7.1 PROCESSING STATES 3

SECTION 7.2 NORMAL PROCESSING STATE 3
- 7.2.1 Instruction Pipeline 3
- 7.2.2 Summary of Pipeline-Related Restrictions 8

SECTION 7.3 EXCEPTION PROCESSING STATE 10
- 7.3.1 Interrupt Types 12
- 7.3.2 Interrupt Priority Structure 12
  - 7.3.2.1 Interrupt Priority Levels 14
  - 7.3.2.2 Exception Priorities Within an IPL 15
- 7.3.3 Interrupt Sources 16
  - 7.3.3.1 Hardware Interrupt Sources 16
  - 7.3.3.2 Software Interrupt Sources 17
  - 7.3.3.3 Other Interrupt Sources 22
- 7.3.4 Interrupt Arbitration 24
- 7.3.5 Interrupt Instruction Fetch 24
- 7.3.6 Instructions Preceding the Interrupt Instruction Fetch 25
- 7.3.7 Interrupt Instruction Execution 26

SECTION 7.4 RESET PROCESSING STATE 33

SECTION 7.5 WAIT PROCESSING STATE 36

SECTION 7.6 STOP PROCESSING STATE 37

7.1 PROCESSING STATES

The DSP56K processor is always in one of five processing states: normal, exception, reset, wait, or stop. This section describes each of the processing states.

7.2 NORMAL PROCESSING STATE

The normal processing state is associated with instruction execution. Details about normal processing of the individual instructions can be found in APPENDIX A - INSTRUCTION SET DETAILS. Instructions are executed using a three-stage pipeline, which is described in the following paragraphs.

7.2.1 Instruction Pipeline

DSP56K instruction execution occurs in a three-stage pipeline, which allows most instructions to execute at a rate of one instruction per instruction cycle. However, certain instructions require additional time to execute: instructions longer than one word, instructions using an addressing mode that requires more than one cycle, and instructions that cause a control-flow change. In the latter case, a cycle is needed to clear the pipeline.

Pipelining allows instruction executions to overlap so that the fetch-decode-execute operations of a given instruction occur concurrently with the fetch-decode-execute operations of other instructions. Specifically, while the processor is executing one instruction, it is decoding the next instruction, and fetching the next instruction from program memory. The processor fetches only one word per cycle, so if an instruction is two words in length, it fetches the additional word before it fetches the next instruction.

Table 7-1 demonstrates pipelining. F1, D1, and E1 refer to the fetch, decode, and execute operations, respectively, of the first instruction. The third instruction, which contains an instruction extension word, takes two instruction cycles to execute. The extension word will be either an absolute address or immediate data. Although it takes three instruction cycles for the pipeline to fill and the first instruction to execute, an instruction usually executes on each instruction cycle thereafter.

Table 7-1 Instruction Pipelining

Operation	Instruction Cycle									
	1	2	3	4	5	6	7	•	•	•
Fetch	F1	F2	F3	F3e	F4	F5	F6	•	•	•
Decode		D1	D2	D3	D3e	D4	D5	•	•	•
Execute			E1	E2	E3	E3e	E4	•	•	•

Each instruction requires a minimum of three instruction cycles (12 clock phases) to be fetched, decoded, and executed. This means that there is a delay of three instruction cycles on powerup to fill the pipe. A new instruction may begin immediately following the previous instruction. Two-word instructions require a minimum of four instruction cycles to execute (three cycles for the first instruction word to move through the pipe and execute and one more cycle for the second word to execute). A new instruction may start after two instruction cycles.

The pipeline is normally transparent to the user. However, there are certain instruction-sequence dependent situations where the pipeline will affect the program execution. Such situations are best described by case studies. Most of these restricted sequences occur because 1) all addresses are formed during instruction decode, or 2) they are the result of contention for an internal resource such as the status register (SR). If the execution of an instruction depends on the relative location of the instruction in a sequence of instructions, there is a pipeline effect. To test for a suspected pipeline effect, compare between the execution of the suspect instruction 1) when it directly follows the previous instruction and 2) when four NOPs are inserted between the two. If there is a difference, it is caused by a pipeline effect. The DSP56K assembler flags instruction sequences with potential pipeline effects so that the user can determine if the operation will execute as expected.

Case 1: The following two examples show similar code sequences.

1. No pipeline effect:

```
ORI #xx,CCR        ;Changes CCR at the end of execution time slot
Jcc xxxx           ;Reads condition codes in SR in its execution time slot
```

The Jcc will test the bits modified by the ORI without any pipeline effect in the code segment above.

2. Instruction that started execution during decode:

```
ORI #04,OMR        ;Sets DE bit at execution time slot
MOVE x:$100,a      ;Reads external RAM instead of internal ROM
```

A pipeline effect occurs in example 2 because the address of the MOVE is formed at its decode time before the ORI changes the DE bit (which changes the memory map) in the ORI's execution time slot. The following code produces the expected results of reading the internal ROM:

```
ORI #04,OMR        ;Sets DE bit at execution time slot
NOP                ;Delays the MOVE so it will read the updated memory map
MOVE x:$100,a      ;Reads internal ROM
```

Case 2: One of the more common sequences where pipeline effects are apparent is as follows:

```
•                   ;Move a number into register Rn (n=0–7).
•
MOVE #xx,Rn
MOVE X:(Rn),A       ;Use the new contents of Rn to address memory.
•
•
```

In this case, before the first MOVE instruction has written Rn during its execution cycle, the second MOVE has accessed the old Rn, using the old contents of Rn. This is because the address for indirect moves is formed during the decode cycle. This overlapping instruction execution in the pipeline causes the pipeline effect. One instruction cycle should be allowed after an address register has been written by a MOVE instruction before the new contents are available for use as an address register by another MOVE instruction. The proper instruction sequence is as follows:

```
•                   ;Move a number into register Rn.
•
MOVE X0,Rn
NOP                 ;Execute any instruction or instruction
•                   ;sequence not using Rn.
•
MOVE X:(Rn),A       Use the new contents of Rn.
```

Case 3: A situation related to Case 2 can be seen in the boot ROM code shown in APPENDIX A of the DSP56001 Technical Data Sheet. At the end of the bootstrap operation, the operation mode register (OMR) is changed to mode #2, and then the program that was loaded is executed. This process is accomplished in the last three instructions:

```
_BOOTEND    MOVEC    #2,OMR     ;Set the operating mode to 2
                                ;(and trigger an exit from
                                ;bootstrap mode).
            ANDI     #$0,CCR    ;Clear SR as if RESET and
                                ;introduce delay needed for
                                ;Op. Mode change.
            JMP      <$0        ;Start fetching from PRAM, P:$0000
```

The JMP instruction generates its jump address during its decode cycle. If the JMP instruction followed the MOVEC, the MOVEC instruction would not have changed the OMR before the JMP instruction formed the fetch address. As a result, the jump would fetch the instruction at P:$0000 of the bootstrap ROM (MOVE #$FFE9,R2). The OMR would then change due to the MOVEC instruction, and the next instruction would be the

second instruction of the downloaded code at P:$0001 of the internal RAM. However, the ANDI instruction allows the OMR to be changed before the JMP instruction uses it, and the JMP fetches P:$0000 of the internal RAM.

Case 4: An interrupt has two additional control cycles that are executed in the interrupt controller concurrently with the fetch, decode, and execute cycles (see Section 7.3 and Figure 7-4). During these two control cycles, the interrupt is arbitrated by comparing the interrupt mask level with the interrupt priority level (IPL) of the interrupt and allowing or disallowing the interrupt. Therefore, if the interrupt mask is changed after an interrupt is arbitrated and accepted as pending but before the interrupt is executed, the interrupt will be executed, regardless of what the mask was changed to. The following examples show that the old interrupt mask is in effect for up to four additional instruction cycles after the interrupt mask is changed. All instructions shown in the examples here are one-word instructions; however, one two-word instruction can replace two one-word instructions except where noted.

1. Program flow with no interrupts after interrupts are disabled:

```
 •
 •
ORI #03,MR      ;Disable interrupts
INST 1
INST 2
INST 3
INST 4
 •
 •
```

2. The four possible variations in program flow that may occur after interrupts are disabled:

```
 •                  •                •              •
 •                  •                •              •
ORI #03,MR          ORI #03,MR       ORI #03,MR     ORI #03,MR
II (See Note 2)     INST 1           INST 1         INST 1
II+1                II               INST 2         INST 2
INST 1              II+1             II             INST 3 (See Note 1)
INST 2              INST 2           II+1           II
INST 3              INST 3           INST 3         II+1
INST 4              INST 4           INST 4         INST 4
 •                  •                •              •
 •                  •                •              •
```

Note 1: INST 3 may be executed at that point only if the preceding instruction (INST 2) was a single-word instruction.

Note 2: II=Interrupt instruction from maskable interrupt.

The following program flow will not occur because the new interrupt mask level becomes effective after a pipeline latency of four instruction cycles:

```
•
•
ORI #03,MR          ;Disable interrupts.
INST 1
INST 2
INST 3
INST 4
II                  ;Interrupts disabled.
II+1                ;Interrupts disabled.
•
•
```

1. Program flow without interrupts after interrupts are re-enabled:

```
•
•
ANDI #00,MR         ;Enable interrupts
INST 1
INST 2
INST 3
INST 4
•
•
```

2. Program flow with interrupts after interrupts are re-enabled:

```
•
•
ANDI #00,MR         ;Enable interrupts
INST 1              ;Uninterruptable
INST 2              ;Uninterruptable
INST 3              ;II fetched
INST 4              ;II+1 fetched
II
II+1
•
•
```

The DO instruction is another instruction that begins execution during the decode cycle of the pipeline. As a result, there are a number of restrictions concerning access contention with the program controller registers accessed by the DO instruction. The ENDDO instruction has similar restrictions. APPENDIX A - INSTRUCTION SET DETAILS contains additional information on the DO and ENDDO instruction restrictions.

Case 5: A resource contention problem can occur when one instruction is using a register during its decode while the instruction executing is accessing the same resource. One example of this is as follows:

```
MOVEC          X:$100,SSH
DO             #$10,END
```

The problem occurs because the MOVEC instruction loads the contents of X:$100 into the system stack high (SSH) during its execution cycle. The DO instruction that follows pushes the stack (LA → SSH, LC → SSL) during its decode cycle. Therefore, the two instructions try writing to the SSH simultaneously and conflict.

7.2.2 Summary of Pipeline-Related Restrictions

The following paragraphs give a summary of the instruction sequences that cause pipeline effects. Additional information about the individual instructions can be found in APPENDIX A - INSTRUCTION SET DETAILS.

DO instruction restrictions:

The DO instruction must not be immediately preceded by any of the following instructions:

BCHG/BCLR/BSET LA, LC, SSH, SSL, or SP
MOVEC/MOVEM to LA, LC, SSH, SSL, or SP
MOVEC/MOVEM from SSH

The DO instruction cannot specify SSH as a source register, as in the following example:

```
DO SSH,xxxx
```

Restrictions near the end of DO loops:

Proper DO loop operation is guaranteed if no instruction starting at address LA-2, LA-1, or LA specifies the program controller registers SR, SP, SSL, LA, LC, or (implicitly) PC as a destination register, or specifies SSH as a source or a destination register.

The restricted instructions at LA-2, LA-1, and LA are as follows:

DO
BCHG/BCLR/BSET LA, LC, SR, SP, SSH, or SSL
BTST SSH
JCLR/JSET/JSCLR/JSSET SSH
MOVEC/MOVEM/MOVEP from SSH
MOVEC/MOVEM/MOVEP to LA, LC, SR, SP, SSH, or SSL
ANDI/ORI MR

The restricted instructions at LA include the following:

Any two-word instruction
Jcc, JMP, JScc, JSR,
REP, RESET, RTI, RTS, STOP, WAIT

Another restriction is shown below:

JSR/JScc/JSCLR/JSSET to LA, if loop flag is set

ENDDO instruction restrictions:

The ENDDO instruction must not be immediately preceded by any of the following instructions:

BCHG/BCLR/BSET LA, LC, SR, SSH, SSL, or SP
MOVEC/MOVEM to LA, LC, SR, SSH, SSL, or SP
MOVEC/MOVEM from SSH
ANDI/ORI MR

RTI and RTS instruction restrictions:

The RTI instruction must not be immediately preceded by any of the following instructions:

BCHG/BCLR/BSET SR, SSH, SSL, or SP
MOVEC/MOVEM to SR, SSH, SSL, or SP
MOVEC/MOVEM from SSH
ANDI MR, ANDI CCR
ORI MR, ORI CCR

The RTS instruction must not be immediately preceded by any of the following instructions:

BCHG/BCLR/BSET SSH, SSL, or SP
MOVEC/MOVEM to SSH, SSL, or SP
MOVEC/MOVEM from SSH

SP and SSH/SSL register manipulation restrictions:

In addition to all the above restrictions concerning SP, SSH, and SSL, the following instruction sequences are illegal:

1. BCHG/BCLR/BSET SP
2. MOVEC/MOVEM/MOVEP from SSH or SSL

and

1. MOVEC/MOVEM to SP
2. MOVEC/MOVEM/MOVEP from SSH or SSL

and

1. MOVEC/MOVEM to SP
2. JCLR/JSET/JSCLR/JSSET SSH or SSL

and

1. BCHG/BCLR/BSET SP
2. JCLR/JSET/JSCLR/JSSET SSH or SSL

Also, the instruction MOVEC SSH,SSH is illegal.

Rn, Nn, and Mn register restrictions:

Due to pipelining, if an address register Rn is the destination of a MOVE-type instruction except MOVEP (MOVE, MOVEC, MOVEM, LUA, Tcc), the new contents will not be available for use as an address pointer until the second following instruction cycle.

Likewise, if an offset register Nn or a modifier register Mn is the destination of a MOVE-type instruction except MOVEP, the new contents will not be available for use in address calculations until the second following instruction cycle.

However, if the processor is in the No Update addressing mode (where Mn and Nn are ignored) and register Mn or Nn is the destination of a MOVE instruction, the next instruction may use the corresponding Rn register as an address pointer. Also, if the processor is in the Postincrement by 1, Postdecrement by 1, or Predecrement by 1 addressing mode (where Nn is ignored), a MOVE to Nn may be immediately followed by an instruction that uses Rn as an address pointer.

Fast interrupt routines:

SWI, STOP, and WAIT may not be used in a fast interrupt routine. (Fast interrupts are described in Section 7.3.1.)

7.3 EXCEPTION PROCESSING STATE (INTERRUPT PROCESSING)

The exception processing state is associated with interrupts that can be generated by conditions inside the DSP or from external sources. In digital signal processing, one of

the main uses of interrupts is to transfer data between DSP memory or registers and a peripheral device. When such an interrupt occurs, a limited context switch with minimal overhead is ideal. A fast interrupt accomplishes a limited context switch. The processor relies on a long interrupt when it must accomplish a more complex task to service the interrupt. Fast interrupts and long interrupts are described in more detail in Section 7.3.1.

There are many sources for interrupts on the DSP56K family of chips, and some of these sources can generate more than one interrupt. The DSP56K family of processors features a prioritized interrupt vector scheme with 32 vectors to provide fast interrupt service. The interrupt priority structure is discussed in Section 7.3.2. The following list outlines how the DSP56K processes interrupts:

1. A hardware interrupt is synchronized with the DSP clock, and the interrupt pending flag for that particular hardware interrupt is set. An interrupt source can have only one interrupt pending at any given time.
2. All pending interrupts (external and internal) are arbitrated to select which interrupt will be processed. The arbiter automatically ignores any interrupts with an IPL lower than the interrupt mask level in the SR and selects the remaining interrupt with the highest IPL.
3. The interrupt controller then freezes the program counter (PC) and fetches two instructions at the two interrupt vector addresses associated with the selected interrupt.
4. The interrupt controller jams the two instructions into the instruction stream and releases the PC, which is used for the next instruction fetch. The next interrupt arbitration then begins.

If neither instruction is a change of program-flow instruction (i.e., a JSR), the state of the machine is not saved on the stack, and a fast interrupt is executed. A long interrupt occurs if one of the interrupt instructions fetched is a JSR instruction. The PC is immediately released, the SR and the PC are saved in the stack, and the jump instruction controls where the next instruction shall be fetched. While either an unconditional jump or a conditional jump can be used to form a long interrupt, they do not store the PC on the stack. Therefore, there is no return path.

Activities 2 and 3 listed above require two additional control cycles, which effectively make the **interrupt** pipeline five levels deep.

7.3.1 Interrupt Types

The DSP56K relies on two types of interrupt routines: fast and long. The fast interrupt

fetches only two words and then automatically resumes execution of the main program; whereas, the long interrupt must be told to return to the main program by executing an RTI instruction. The fast routine consists of two automatically inserted interrupt instruction words. These words can contain any unrestricted, single two-word instruction or any two one-word instructions (see Section A.9 in APPENDIX A - INSTRUCTION SET DETAILS for a list of restrictions). Fast interrupt routines are never interruptible.

CAUTION

Status is not preserved during a fast interrupt routine; therefore, instructions that modify status should not be used at the interrupt starting address and interrupt starting address +1.

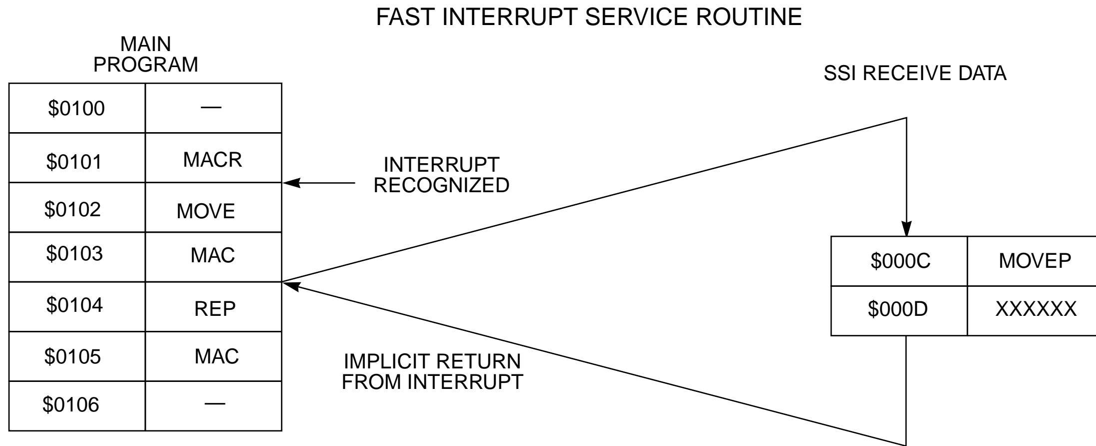
If one of the instructions in the fast routine is a JSR, then a long interrupt routine is formed. The following actions occur during execution of the JSR instruction when it occurs in the interrupt starting address or in the interrupt starting address +1:

1. The PC (containing the return address) and the SR are stacked.
2. The loop flag is reset.
3. The scaling mode bits are reset.
4. The IPL is raised to disallow further interrupts at the same or lower levels (except that hardware $\overline{\text{RESET}}$, NMI, stack error, trace, and SWI can always interrupt).
5. The trace bit in the SR is cleared (in the DSP56000/56001 only).

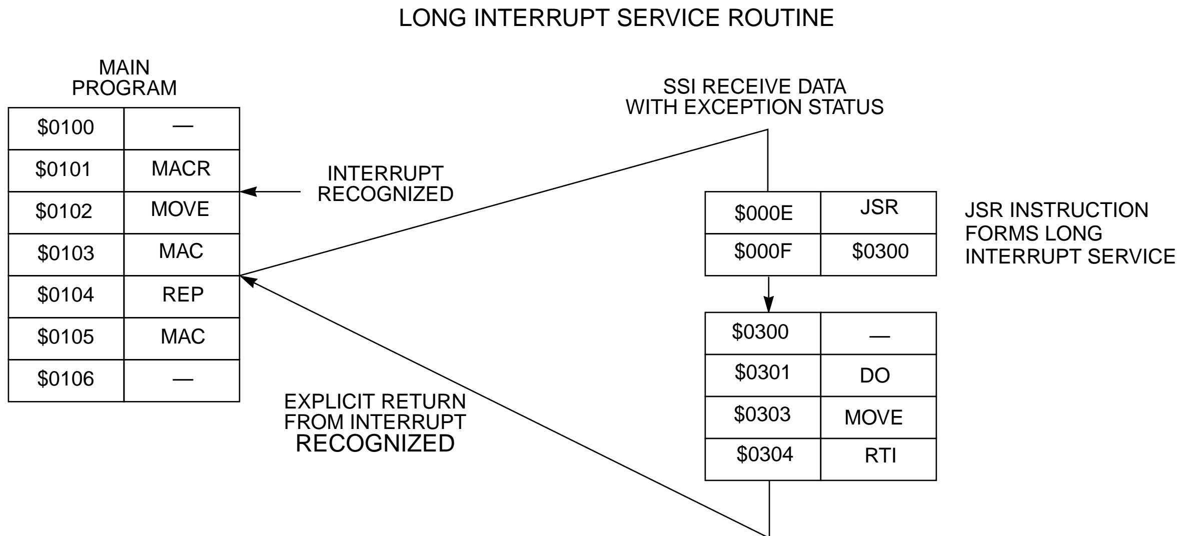
The long interrupt routine should be terminated by an RTI. Long interrupt routines are interruptible by higher priority interrupts. Figure 7-1 shows examples of fast and long interrupts.

7.3.2 Interrupt Priority Structure

Interrupts are organized in a flexible priority structure. Each interrupt has an associated interrupt priority level (IPL) that can range from zero to three. Levels 0 (lowest level), 1, and 2 are maskable. Level 3 is the highest IPL and is not maskable. The only IPL 3 interrupts are $\overline{\text{RESET}}$, illegal instruction interrupt (III), nonmaskable interrupt ($\overline{\text{NMI}}$), stack error, trace, and software interrupt (SWI). The interrupt mask bits (I1, I0) in the SR reflect the current priority level and indicate the IPL needed for an interrupt source to interrupt the processor (see Table 7-2). Interrupts are inhibited for all priority levels below the current processor priority level. However, level 3 interrupts are not maskable and therefore can always interrupt the processor. DSP56K Family central processor interrupt sources



(a) DSP56K Fast Interrupt



(b) DSP56K Long Interrupt

Figure 7-1 Fast and Long Interrupt Examples

and their IPLs are listed in Table 7-6. For information on on-chip peripheral interrupt pri-

Table 7-2 Status Register Interrupt Mask Bits

I1	I0	Exceptions Permitted	Exceptions Masked
0	0	IPL 0, 1, 2, 3	None
0	1	IPL 1, 2, 3	IPL 0
1	0	IPL 2, 3	IPL 0, 1
1	1	IPL 3	IPL 0, 1, 2

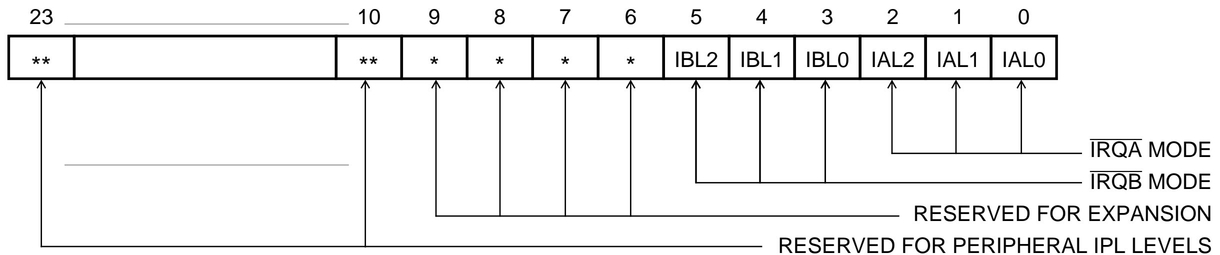
ority levels, see the individual DSP56K family member's User's Manual.

7.3.2.1 Interrupt Priority Levels

The IPL for each on-chip peripheral device (HI, SSI, SCI) and for each external interrupt source ($\overline{\text{IRQA}}$, $\overline{\text{IRQB}}$) can be programmed to one of the three maskable priority levels (IPL 0, 1, or 2) under software control. IPLs are set by writing to the interrupt priority register shown in Figure 7-2. This read/write register is located in program memory at address $FFFF. It specifies the IPL for each of the interrupting devices including IRQA, IRQB and each peripheral device. (For specific peripheral information, see the specific DSP56K family member's User's Manual.) In addition, it specifies the trigger mode of the external interrupt sources and is used to enable or disable the individual external interrupts. The interrupt priority register is cleared on $\overline{\text{RESET}}$ or by the reset instruction. Table 7-3 defines the IPL bits. Table 7-4 defines the external interrupt trigger mode bits.

7.3.2.2 Exception Priorities Within an IPL

If more than one interrupt is pending when an instruction is executed, the processor will service the interrupt with the highest priority level first. When multiple interrupt requests



Bits 6 to 9 are reserved, read as zero and should be written with zero for future compatibility.

Figure 7-2 Interrupt Priority Register (Addr X:$FFFF)

Table 7-3 Interrupt Priority Level Bits

Table 7-4 External Interrupt

xxL1	xxL0	Enabled	IPL
0	0	No	—

with the same IPL are pending, a second fixed-priority structure within that IPL determines which interrupt the processor will service. The fixed priority of interrupts within an IPL and the interrupt enable bits for all interrupts are shown in Table 7-5.

7.3.3 Interrupt Sources

Interrupts can originate from any of the vector addresses listed in Table 7-6, which shows the corresponding interrupt starting address for each interrupt source. These addresses are located in the first 64 locations of program memory.

Table 7-5 Central Processor Interrupt Priorities Within an IPL

Priority	Exception	Enabled By	Bit No.	X Data Memory Address
Level 3 (Nonmaskable)				
Highest	Hardware $\overline{\text{RESET}}$	—	—	—
	III	—	—	—
	NMI	—	—	—
	Stack Error	—	—	—
	Trace	—	—	—
Lowest	SWI	—	—	—
Levels 0, 1, 2 (Maskable)				
Higher	$\overline{\text{IRQA}}$ (External Interrupt)	$\overline{\text{IRQA}}$ Mode Bits	0 and 1	$FFFF
Lower	$\overline{\text{IRQB}}$ (External Interrupt)	$\overline{\text{IRQB}}$ Mode Bits	3 and 4	$FFFF

Table 7-6 Interrupt Sources

Interrupt Starting Address	IPL	Interrupt Source
$0000	3	Hardware RESET
$0002	3	Stack Error
$0004	3	Trace
$0006	3	SWI
$0008	0 - 2	IRQA
$000A	0 - 2	IRQB
:	:	Vectors available for peripherals
$001E	3	NMI
:	:	Vectors available for peripherals
$003E	3	Illegal Instruction

When an interrupt occurs, the instruction at the interrupt starting address is fetched first. Because the program flow is directed to a different starting address for each interrupt, the interrupt structure of the DSP56K can be described as “vectored”. A vectored interrupt structure has low execution overhead. If it is known beforehand that certain interrupts will not be used, those interrupt vector locations can be used for program or data storage.

7.3.3.1 Hardware Interrupt Sources

There are two types of hardware interrupts in the DSP56K: internal and external. The internal interrupt sources include all of the on-chip peripheral devices. For further information on a device’s internal interrupt sources, see the device’s individual User’s Manual.

The external hardware interrupt sources are the RESET, NMI, IRQA, and IRQB pins on the program interrupt controller in the Program Control Unit.

The level sensitive RESET interrupt is the highest priority interrupt with an IPL of 3. IRQA and IRQB can be programmed to one of three priority levels: 0, 1, or 2 - all of which are maskable. IRQA and IRQB have independent enable control and can be programmed to be level sensitive or edge sensitive. Since level-sensitive interrupts will not be cleared automatically when they are serviced, they must be cleared by other means to prevent multiple interrupts. Edge-sensitive interrupts are latched as pending on the high-to-low transition of the interrupt input and are automatically cleared when the interrupt is serviced.

When either the $\overline{\text{IRQA}}$ or $\overline{\text{IRQB}}$ pin is disabled in the interrupt priority register, the interrupt request coming in on the pin will be ignored, regardless of whether the input was defined as level sensitive or edge sensitive. If the interrupt input is defined as edge sensitive, its edge-detection latch will remain in the reset state for as long as the interrupt pin is disabled. If the interrupt is defined as level-sensitive, its edge-detection latch will stay in the reset state. If the level-sensitive interrupt is disabled while it is pending it will be cancelled. However, if the interrupt has been fetched, it normally will not be cancelled.

The processor begins interrupt service by fetching the instruction word in the first vector location. The interrupt is considered finished when the processor fetches the instruction word in the second vector location.

In an edge-triggered interrupt, the internal latch is automatically cleared when the second vector location is fetched. The fetch of the first vector location does not guarantee that the second location will be fetched. Figure 7-3 illustrates the one case where the second vector location is not fetched. The SWI instruction in the figure discards the fetch of the first interrupt vector to ensure that the SWI vectors will be fetched. Instruction n4 is decoded as an SWI while ii1 is being fetched. Execution of the SWI requires that ii1 be discarded and the two SWI vectors (ii3 and ii4) be fetched instead.

INTERRUPT CONTROL CYCLE 1		i		i*							
INTERRUPT CONTROL CYCLE 2			i		i*						
FETCH	n3	n4	n5	ii1		ii3	ii4	sw1	sw2	sw3	sw4
DECODE	n2	n3	SWI	—	—	—	JSR	—	sw1	sw2	sw3
EXECUTE	n1	n2	n3	SWI	NOP	NOP	NOP	JSR	—	sw1	sw2
INSTRUCTION BEING DECODED	1										

i = INTERRUPT REQUEST
i* = INTERRUPT REQUEST GENERATED BY SWI
ii1 = FIRST VECTOR OF INTERRUPT i
ii3 = FIRST SWI VECTOR (ONE-WORD JSR)
ii4 = SECOND SWI VECTOR
n = NORMAL INSTRUCTION WORD
n4 = SWI
sw = INSTRUCTIONS PERTAINING TO THE SWI LONG INTERRUPT ROUTINE

Figure 7-3 Interrupting an SWI

CAUTION

On all level-sensitive interrupts, the interrupt must be externally released before interrupts are internally re-enabled. Otherwise, the processor will be interrupted repeatedly until the release of the level-sensitive interrupt occurs.

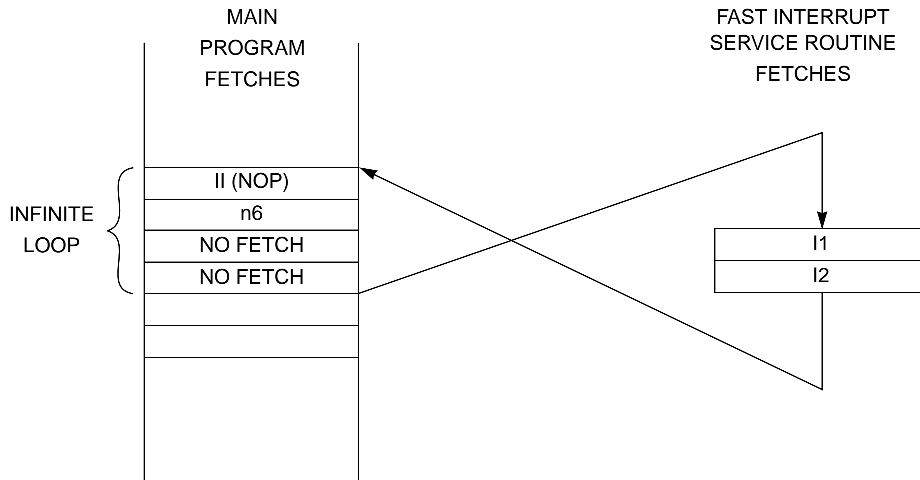
The edge sensitive NMI is a priority 3 interrupt and cannot be masked. Only $\overline{\text{RESET}}$ and illegal instruction have higher priority than NMI.

7.3.3.2 Software Interrupt Sources

There are two software interrupt sources — software interrupt (SWI) and illegal instruction interrupt (III).

SWI is a nonmaskable interrupt (IPL 3), which is serviced immediately following the SWI instruction execution, usually using a long interrupt service routine. The difference between an SWI and a JSR instruction is that the SWI sets the interrupt mask to prevent interrupts below IPL 3 from being serviced. The SWI’s ability to mask out lower level interrupts makes it very useful for setting breakpoints in monitor programs. The JSR instruction does not affect the interrupt mask.

The III is also a nonmaskable interrupt (IPL 3). It is serviced immediately following the execution or the attempted execution of an illegal instruction (any undefined operation code). IIIs are fatal errors. Only a long interrupt routine should be used for the III routine. RTI or RTS should not be used at the end of the interrupt routine because, during the III service, the JSR located in the III vector will normally stack the address of the illegal instruction (see Figure 7-4). Returning from the interrupt routine would cause the processor to attempt to execute the illegal interrupt again and cause an infinite loop which can only be broken by cycling power. This long interrupt (see Figure 7-4) can be used as a diagnostic tool to allow the programmer to examine the stack (MOVE SSH, dest) and locate the illegal instruction, or the application program can be restarted with the hope that the failure was a soft error. The illegal instruction is useful for triggering the illegal interrupt service routine to see if the III routine can recover from illegal instructions.



(a) Instruction Fetches from Memory

ILLEGAL INSTRUCTION INTERRUPT
RECOGNIZED AS PENDING

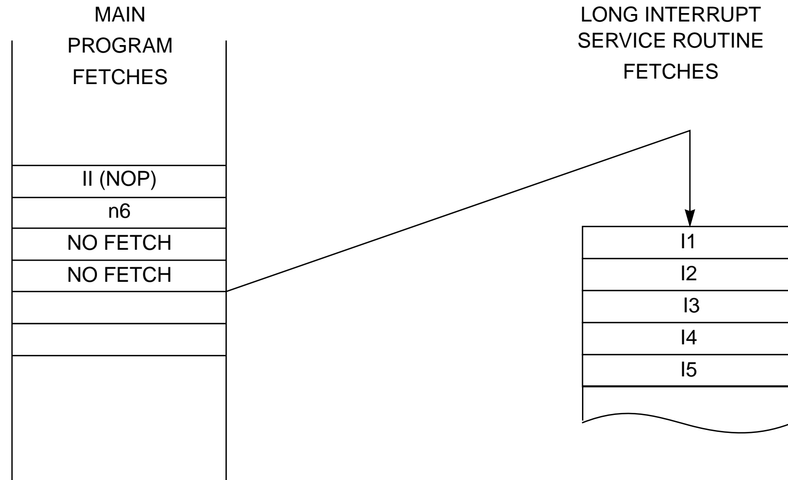
ILLEGAL INSTRUCTION INTERRUPT
RECOGNIZED AS PENDING

INTERRUPT CONTROL CYCLE 1								i						
INTERRUPT CONTROL CYCLE 2									i					
FETCH		n1	n2	n3	n4	n5	n6	—	—	ii1	ii2	n5		
DECODE			n1	n2	n3	n4	II	—	—	—	ii1	ii2	II	
EXECUTE				n1	n2	n3	n4	NOP	—	—	—	ii1	ii2	NOP
INSTRUCTION CYCLE COUNT	1	2	3	4	5	6	7	8	9	10	11	12	13	14

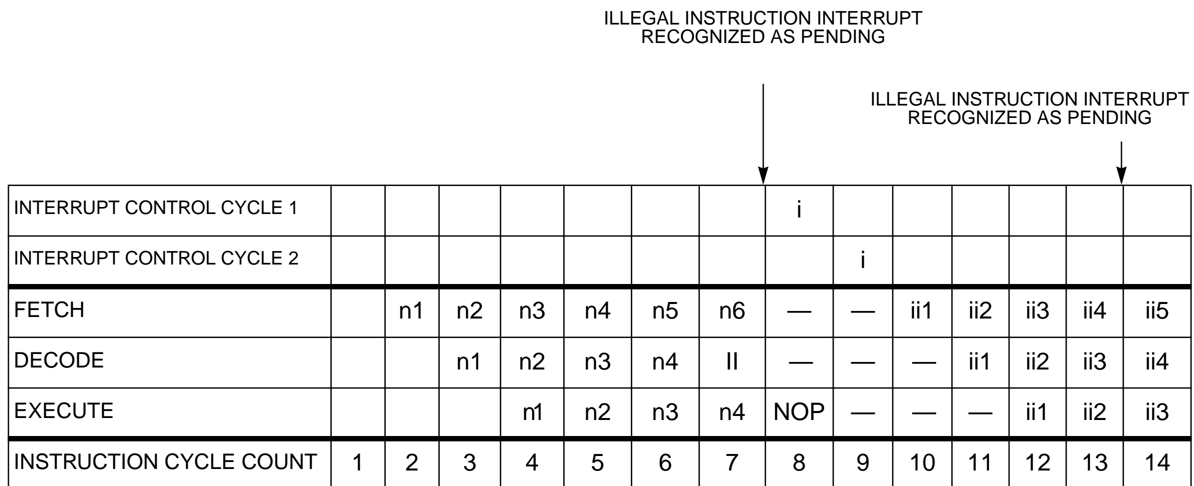
i = INTERRUPT
ii = INTERRUPT INSTRUCTION WORD
II = ILLEGAL INSTRUCTION
n = NORMAL INSTRUCTION WORD

(b) Program Controller Pipeline

Figure 7-4 Illegal Instruction Interrupt Serviced by a Fast Interrupt



(a) Instruction Fetches from Memory

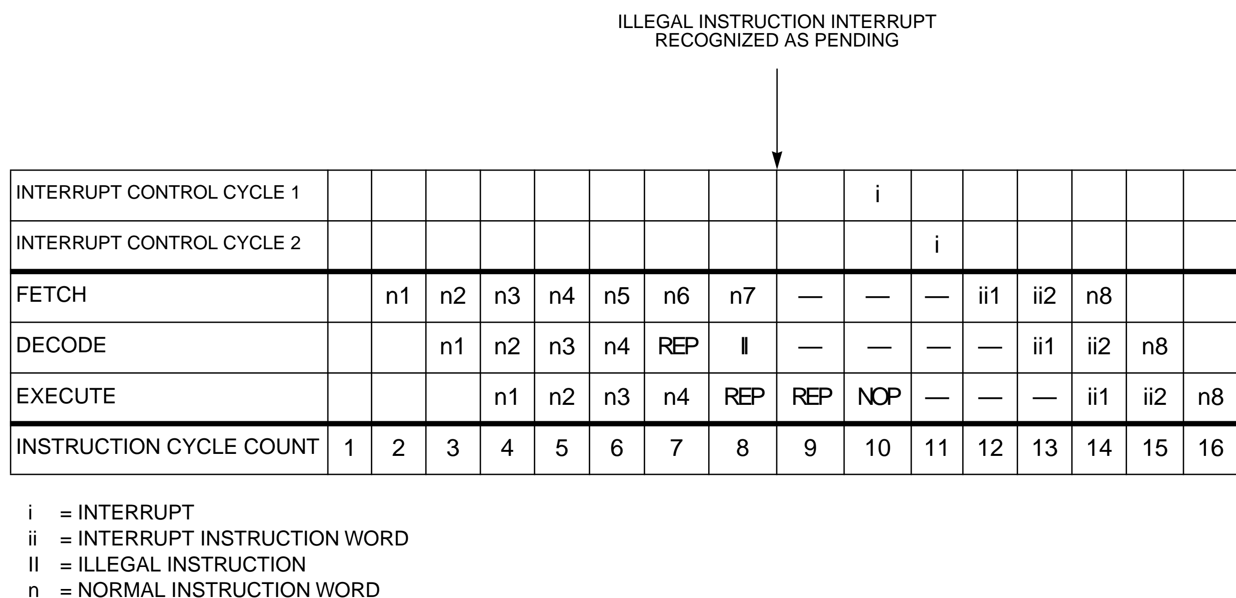


INTERRUPT CONTROL CYCLE 1								i						
INTERRUPT CONTROL CYCLE 2									i					
FETCH		n1	n2	n3	n4	n5	n6	—	—	ii1	ii2	ii3	ii4	ii5
DECODE			n1	n2	n3	n4	II	—	—	—	ii1	ii2	ii3	ii4
EXECUTE				n1	n2	n3	n4	NOP	—	—	—	ii1	ii2	ii3
INSTRUCTION CYCLE COUNT	1	2	3	4	5	6	7	8	9	10	11	12	13	14

i = INTERRUPT
ii = INTERRUPT INSTRUCTION WORD
II = ILLEGAL INSTRUCTION
n = NORMAL INSTRUCTION WORD

(b) Program Controller Pipeline

Figure 7-5 Illegal Instruction Interrupt Serviced by a Long Interrupt



INTERRUPT CONTROL CYCLE 1										i						
INTERRUPT CONTROL CYCLE 2											i					
FETCH		n1	n2	n3	n4	n5	n6	n7	—	—	—	ii1	ii2	n8		
DECODE			n1	n2	n3	n4	REP	II	—	—	—	—	ii1	ii2	n8	
EXECUTE				n1	n2	n3	n4	REP	REP	NOP	—	—	—	ii1	ii2	n8
INSTRUCTION CYCLE COUNT	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

i = INTERRUPT
ii = INTERRUPT INSTRUCTION WORD
II = ILLEGAL INSTRUCTION
n = NORMAL INSTRUCTION WORD

Figure 7-6 Repeated Illegal Instruction

There are two cases in which the stacked address will not point to the illegal instruction:

1. If the illegal instruction is one of the two instructions at an interrupt vector location and is fetched during a regular interrupt service, the processor will stack the address of the next sequential instruction in the normal instruction flow (the regular return address of the interrupt routine that had the illegal opcode in its vector).
2. If the illegal instruction follows an REP instruction (see Figure 7-6), the processor will effectively execute the illegal instruction as a repeated NOP and the interrupt vector will then be inserted in the pipeline. The next instruction will be fetched but will not be decoded or executed. The processor will stack the address of the next sequential instruction, which is two instructions after the illegal instruction.

In DO loops, if the illegal instruction is in the loop address (LA) location and the instruction preceding it (i.e., at LA-1) is being interrupted, the loop counter (LC) will be decremented as if the loop had reached the LA instruction. When the interrupt service ends and the instruction flow returns to the loop, the illegal instruction will be refetched (since it is the next sequential instruction in the flow). The loop state machine will again decrement LC because the LA instruction is being executed. At this point, the illegal instruction will trigger the III. The result is that the loop state machine decrements LC twice in one loop due to the presence of the illegal opcode at the LA location.

7.3.3.3 Other Interrupt Sources

Other interrupt sources include the stack error interrupt and trace interrupt (DSP56000/56001) which are IPL3 interrupts.

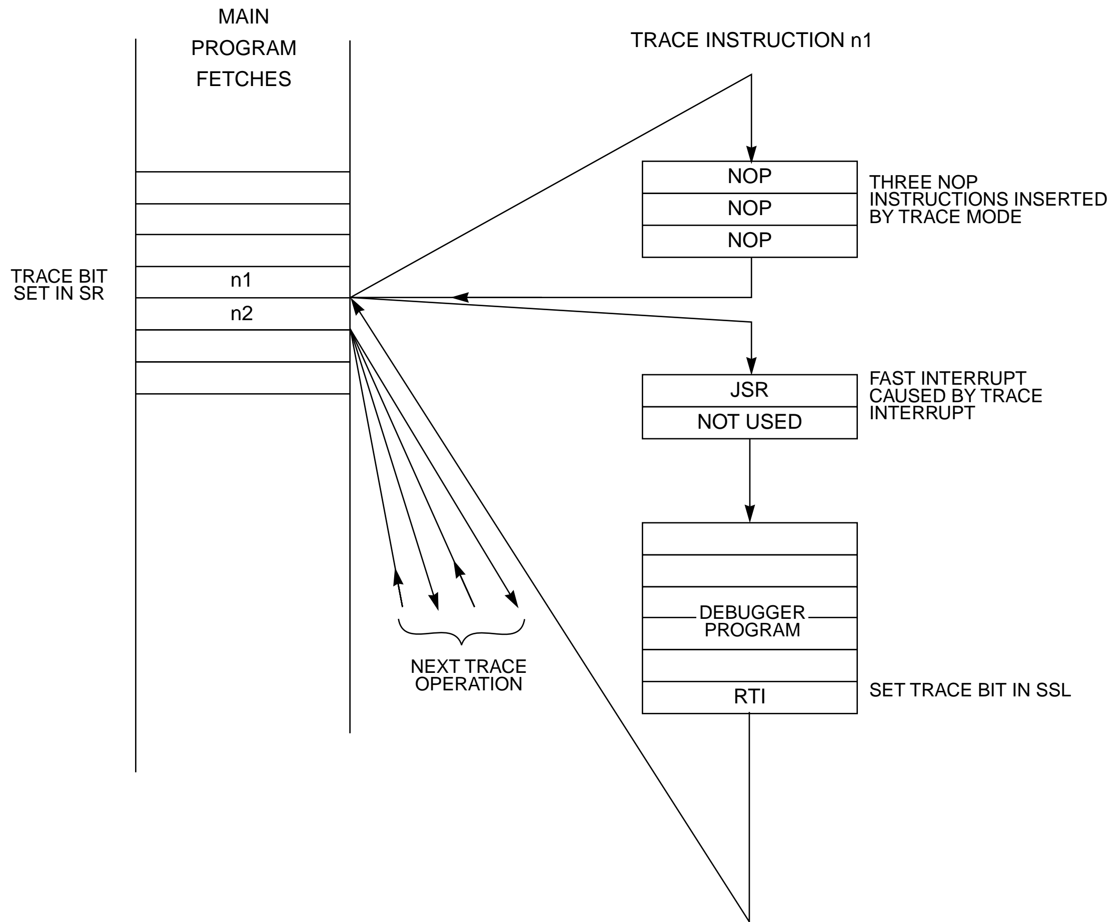
An overflow or underflow of the system stack (SS) causes a stack error interrupt which is vectored to P:$0002 (see SECTION 5 - PROGRAM CONTROL UNIT for additional information on the stack error flag). Since the stack error is nonrecoverable, a long interrupt should be used to service it. The service routine should not end in an RTI because executing an RTI instruction “pops” the stack, which has been corrupted.

The DSP56000/56001 includes a facility for instruction-by-instruction tracing as a program development aid. This trace mode generates a trace exception after each instruction executed (see Figure 7-7), which can be used by a debugger program to monitor the execution of a program. (With members of the DSP56K family other than DSP56000/56001, use the OnCE trace mode described in 10.5.)

The trace bit in the SR defines the trace mode. In the trace mode, the processor will generate a trace exception after it executes each instruction. When the processor is servicing the trace exception, it expects to encounter a JSR in the trace vector locations, thereby forming a long interrupt routine. The JSR stacks the SR and clears the trace bit to prevent tracing while executing the trace exception service routine. This service routine should end with an RTI instruction, which restores the SR (with the trace bit set) from the SS, and causes the next instruction to be traced. The pipeline must be flushed to allow each sequential instruction to be traced. The tracing facility appends three instruction cycles to the end of each instruction traced (see the three NOP instructions shown in Figure 7-7) to flush the pipeline and allow the next trace interrupt to follow the next sequential interrupt.

During tracing, the processor considers the REP instruction and the instruction being repeated as a single two-word instruction. That is, only after executing the REP instruction and all of the repeats of the next instruction will the trace exception be generated.

Fast interrupts can not be traced because they are uninterruptable. Long interrupts will not be traced unless the processor enters the trace mode in the subroutine because the SR is pushed on the stack and the trace bit is cleared. Tracing is resumed upon returning from a long interrupt because the trace bit is restored when the SR is restored. Interrupts are not likely to occur during tracing because only an interrupt with a higher IPL can interrupt during a trace operation. While executing the program being traced, the trace interrupt will always be pending and will win the interrupt arbitration. During the trace interrupt, the interrupt mask is set to reject interrupts below IPL3.



(a) Instruction Fetches from Memory

INTERRUPT SYNCHRONIZED AND RECOGNIZED AS PENDING (at cycle 1)

INTERRUPT SYNCHRONIZED AND RECOGNIZED AS PENDING (at cycle 13)

INTERRUPT CONTROL CYCLE 1	i												i					
INTERRUPT CONTROL CYCLE 2		i													i			
FETCH		n1	NOP	NOP	NOP	JSR	—	TRACE PROGRAM			RTI	—	n2	NOP	NOP	NOP		
DECODE			n1	NOP	NOP	NOP	JSR	NOP	TRACE PROGRAM			RTI	NOP	n2	NOP	NOP	NOP	
EXECUTE				n1	NOP	NOP	NOP	JSR	NOP	TRACE PROGRAM			RTI	NOP	n2	NOP	NOP	NOP
INSTRUCTION CYCLE COUNT	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18

i = INTERRUPT
ii = INTERRUPT INSTRUCTION WORD
II = ILLEGAL INSTRUCTION
n = NORMAL INSTRUCTION WORD

(b) Program Controller Pipeline

Figure 7-7 Trace Exception

7.3.4 Interrupt Arbitration

Interrupt arbitration and control, which occurs concurrently with the fetch-decode-execute cycle, takes two instruction cycles. External interrupts are internally synchronized with the processor clock before their interrupt-pending flags are set. Each external and internal interrupt has its own flag. After each instruction is executed, the DSP arbitrates all interrupts. During arbitration, each interrupt's IPL is compared with the interrupt mask in the SR, and the interrupt is either allowed or disallowed. The remaining interrupts are prioritized according to the IPLs shown in Table 7-5, and the highest priority interrupt is chosen. The interrupt vector is then calculated so that the program interrupt controller can fetch the first interrupt instruction.

Interrupts from a given source are not buffered. The processor won't arbitrate a new interrupt from the same source until after it fetches the second interrupt vector of the current interrupt.

The internal interrupt acknowledge signal clears the edge-triggered interrupt flags and the internal latches of the NMI, SWI, and trace interrupts. The stack error bit in the stack pointer register is "sticky" and requires a "MOVE" or a bit clear operation directly on the stack pointer register. Some peripheral interrupts may also be cleared by the internal interrupt acknowledge signal, as defined in their specifications. Peripheral interrupt requests that need a read/write action to some register do not receive the internal interrupt acknowledge signal, and they will remain pending until their registers are read/written. Further, level-triggered interrupts will not be cleared. The acknowledge signal will be generated after the interrupt vectors have been generated, not before.

7.3.5 Interrupt Instruction Fetch

The interrupt controller generates an interrupt instruction fetch address, which points to the first instruction word of a two-word interrupt routine. This address is used for the next instruction fetch, instead of the contents of the PC, and the interrupt instruction fetch address +1 is used for the subsequent instruction fetch. While the interrupt instructions are being fetched, the PC cannot be updated. After the two interrupt words have been fetched, the PC is used for any subsequent instruction fetches.

After both interrupt vectors have been fetched, they are guaranteed to be executed. This is true even if the instruction that is currently being executed is a change-of-flow instruction (i.e., JMP, JSR, etc.) that would normally ignore the instructions in the pipe. After the interrupt instruction fetch, the PC will point to the instruction that would have been fetched if the interrupt instructions had not been inserted.

7.3.6 Instructions Preceding the Interrupt Instruction Fetch

The following one-word instructions are aborted when they are fetched in the cycle preceding the fetch of the first interrupt instruction word — REP, STOP, WAIT, RESET, RTI, RTS, Jcc, JMP, JScc, and JSR.

Two-word instructions are aborted when the first interrupt instruction word fetched will replace the fetch of the second word of the two-word instruction. Aborted instructions are refetched when program control returns from the interrupt routine. The PC is adjusted appropriately before the end of the decode cycle of the aborted instruction.

If the first interrupt word fetch occurs in the cycle following the fetch of a one-word instruction not previously listed or the second word of a two-word instruction, that instruction will complete normally before the start of the interrupt routine.

The following cases have been identified where service of an interrupt might encounter an extra delay:

1. If a long interrupt routine is used to service an SWI, then the processor priority level is set to 3. Thus, all interrupts except other level-3 interrupts are disabled until the SWI service routine terminates with an RTI (unless the SWI service routine software lowers the processor priority level).
2. While servicing an interrupt, the next interrupt service will be delayed according to the following rule: after the first interrupt instruction word reaches the instruction decoder, at least three more instructions will be decoded before decoding the next first interrupt instruction word. If any one pair of instructions being counted is the REP instruction followed by an instruction to be repeated, then the combination is counted as two instructions independent of the number of repeats done. Sequential REP combinations will cause pending interrupts to be rejected and can not be interrupted until the sequence of REP combinations ends.
3. The following instructions are not interruptible: SWI, STOP, WAIT, and RESET.
4. The REP instruction and the instruction being repeated are not interruptible.
5. If the trace bit in the SR (DSP56000/56001 only) is set, the only interrupts that will be processed are the hardware RESET, III,NMI, stack error, and trace. Peripheral and external interrupt requests will be ignored. The interrupt generated by the SWI instruction will be ignored.

7.3.7 Interrupt Instruction Execution

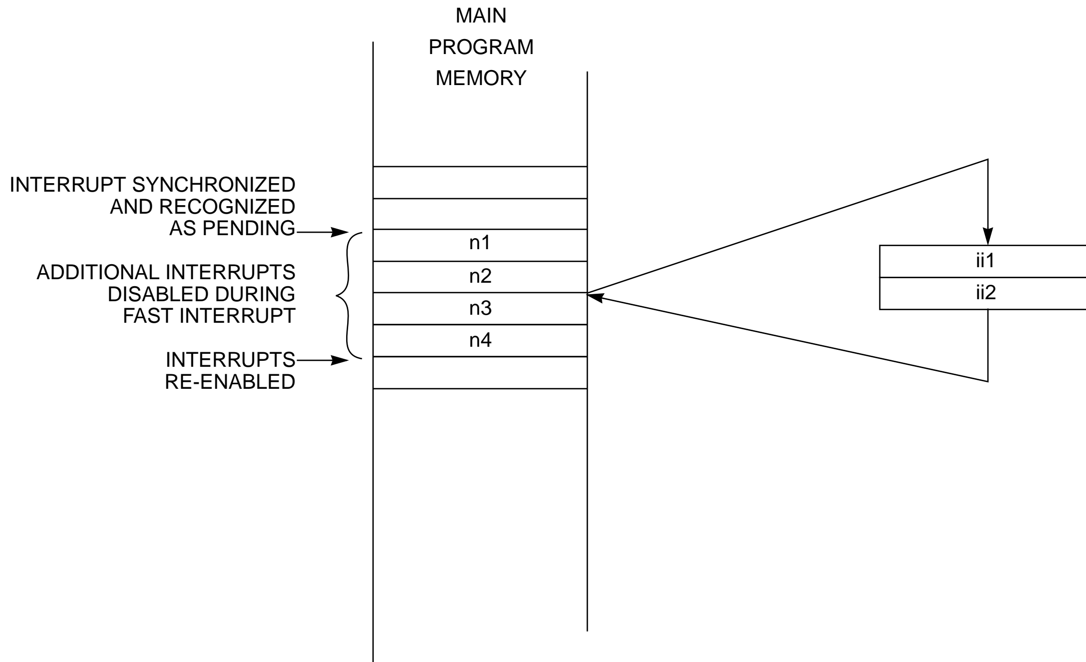
Interrupt instruction execution is considered “fast” if neither of the instructions of the interrupt service routine causes a change of flow. A JSR within a fast interrupt routine forms a long interrupt, which is terminated with an RTI instruction to restore the PC and SR from the stack and return to normal program execution. Reset is a special exception, which will normally contain only a JMP instruction at the exception start address. At the programmer’s option, almost any instruction can be used in the fast interrupt routine. The restricted instructions include SWI, STOP, and WAIT. Figure 7-8 and Figure 7-10 show the fast and the long interrupt service routines. The fast interrupt executes only two instructions and then automatically resumes execution of the main program; whereas, the long interrupt must be told to return to the main program by executing an RTI instruction.

Figure 7-8 illustrates the effect of a fast interrupt routine in the stream of instruction fetches.

Figure 7-9 shows the sequence of instruction decodes between two fast interrupts. Four decodes occur between the two interrupt decodes (two after the first interrupt and two preceding the second interrupt). The requirement for these four decodes establishes the maximum rate at which the DSP56K will respond to interrupts — namely, one interrupt every six instructions (six instruction cycles if all six instructions are one instruction cycle each). Since some instructions take more than one instruction cycle, the minimum number of instructions between two interrupts may be more than six instruction cycles.

The execution of a fast interrupt routine always conforms to the following rules:

1. A JSR to the starting address of the interrupt service routine is not located at one of the two interrupt vector addresses.
2. The processor status is not saved.
3. The fast interrupt routine may (but should not) modify the status of the normal instruction stream.
4. The fast interrupt routine may contain any single two-word instruction or any two one-word instructions except SWI, STOP, and WAIT.
5. The PC, which contains the address of the next instruction to be executed in normal processing remains unchanged during a fast interrupt routine.



ii = INTERRUPT INSTRUCTION
n = NORMAL INSTRUCTION

(a) Instruction Fetches from Memory

INTERRUPT SYNCHRONIZED AND RECOGNIZED AS PENDING

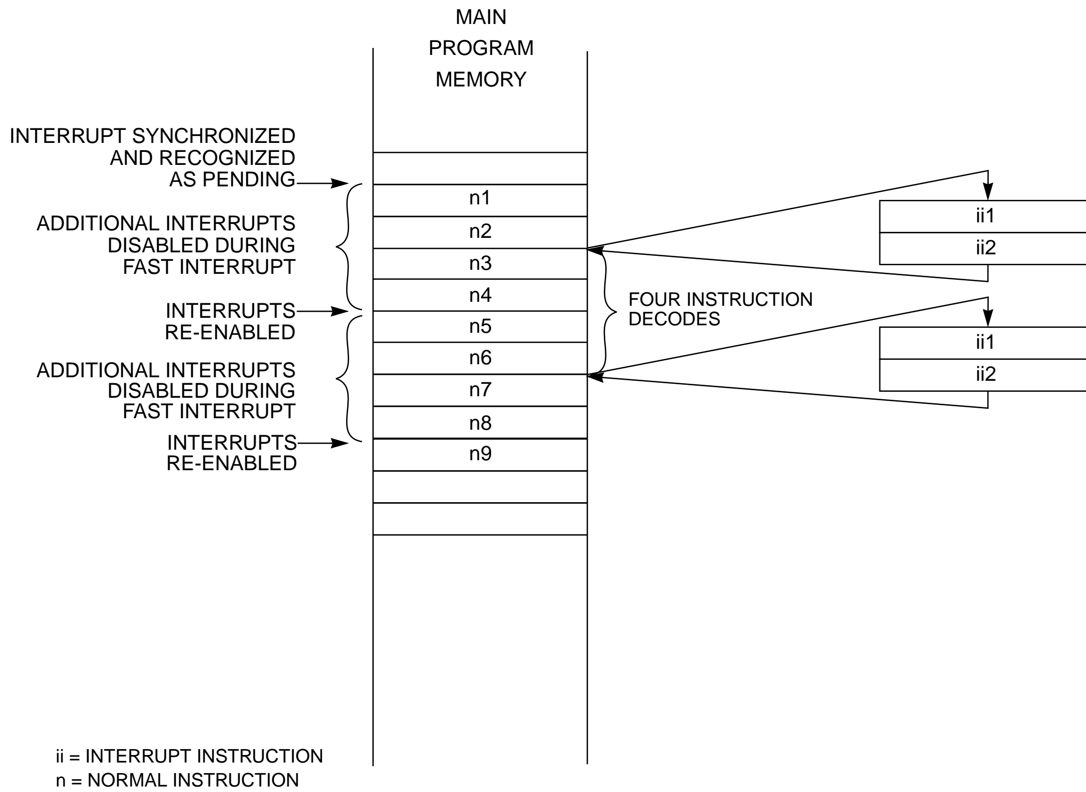
INTERRUPTS RE-ENABLED

INTERRUPT CONTROL CYCLE 1	i							
INTERRUPT CONTROL CYCLE 2		i						
FETCH	n1	n2	ii1	ii2	n3	n4		
DECODE		n1	n2	ii1	ii2	n3	n4	
EXECUTE			n1	n2	ii1	ii2	n3	n4
INSTRUCTION CYCLE COUNT	1	2	3	4	5	6	7	8

i = INTERRUPT
ii = INTERRUPT INSTRUCTION WORD
n = NORMAL INSTRUCTION WORD

(b) Program Controller Pipeline

Figure 7-8 Fast Interrupt Service Routine



(a) Instruction Fetches from Memory

INTERRUPT SYNCHRONIZED AND RECOGNIZED AS PENDING

INTERRUPTS RE-ENABLED

6 I_{cyc}

INTERRUPT CONTROL CYCLE 1	i						i					
INTERRUPT CONTROL CYCLE 2		i						i				
FETCH	n1	n2	ii1	ii2	n3	n4	n5	n6	ii1	ii2		
DECODE		n1	n2	ii1	ii2	n3	n4	n5	n6	ii1	ii2	
EXECUTE			n1	n2	ii1	ii2	n3	n4	n5	n6	ii1	ii2
INSTRUCTION CYCLE COUNT	1	2	3	4	5	6	7	8	9	10	11	12

i = INTERRUPT
ii = INTERRUPT INSTRUCTION WORD
n = NORMAL INSTRUCTION WORD

(b) Program Controller Pipeline

Figure 7-9 Two Consecutive Fast Interrupts

6. The fast interrupt returns without an RTI.
7. Normal instruction fetching resumes using the PC following the completion of the fast interrupt routine.
8. A fast interrupt is not interruptible.
9. A JSR instruction within the fast interrupt routine forms a long interrupt routine.
10. The primary application is to move data between memory and I/O devices.

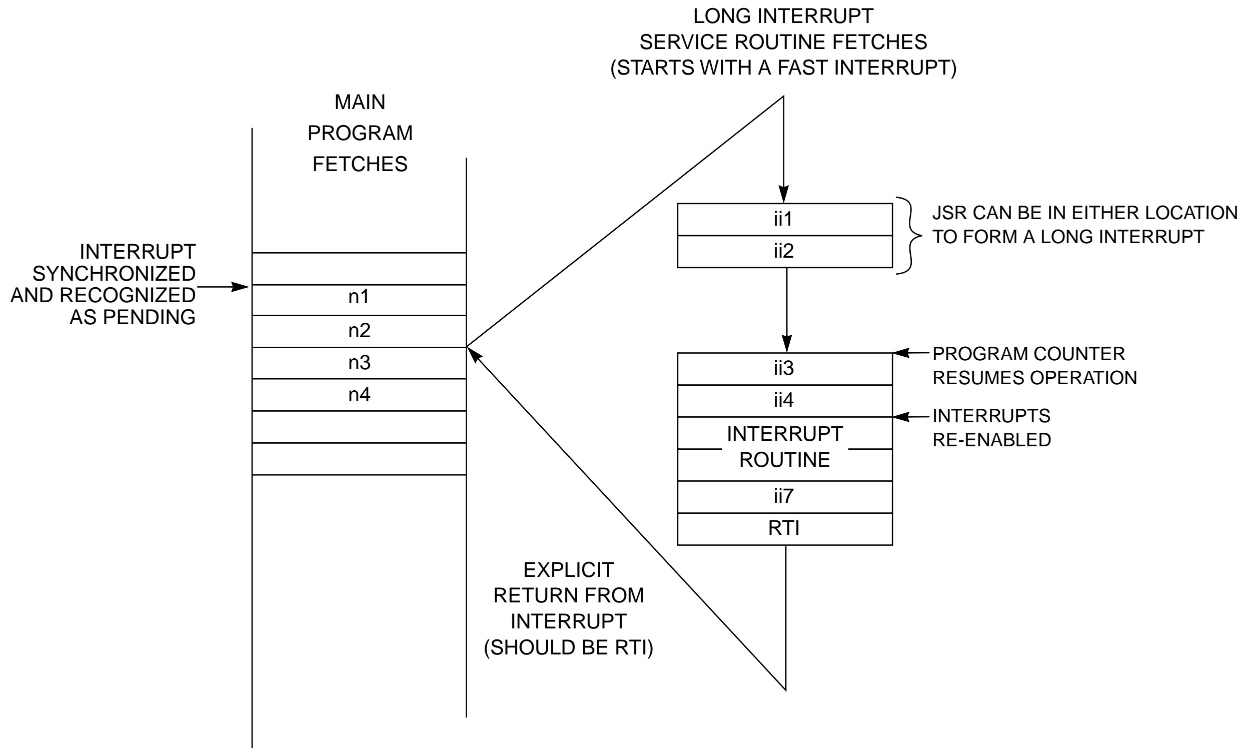
The execution of a long interrupt routine always conforms to the following rules:

1. A JSR to the starting address of the interrupt service routine is located at one of the two interrupt vector addresses.
2. During execution of the JSR instruction, the PC and SR are stacked. The interrupt mask bits of the SR are updated to mask interrupts of the same or lower priority. The loop flag, trace bit, double precision multiply mode bit, and scaling mode bits are reset.
3. The first instruction word of the next interrupt service (of higher IPL) will reach the decoder only after the decoding of at least four instructions following the decoding of the first instruction of the previous interrupt.
4. The interrupt service routine can be interrupted — i.e., nested interrupts are supported.
5. The long interrupt routine, which can be any length, should be terminated by an RTI, which restores the PC and SR from the stack.

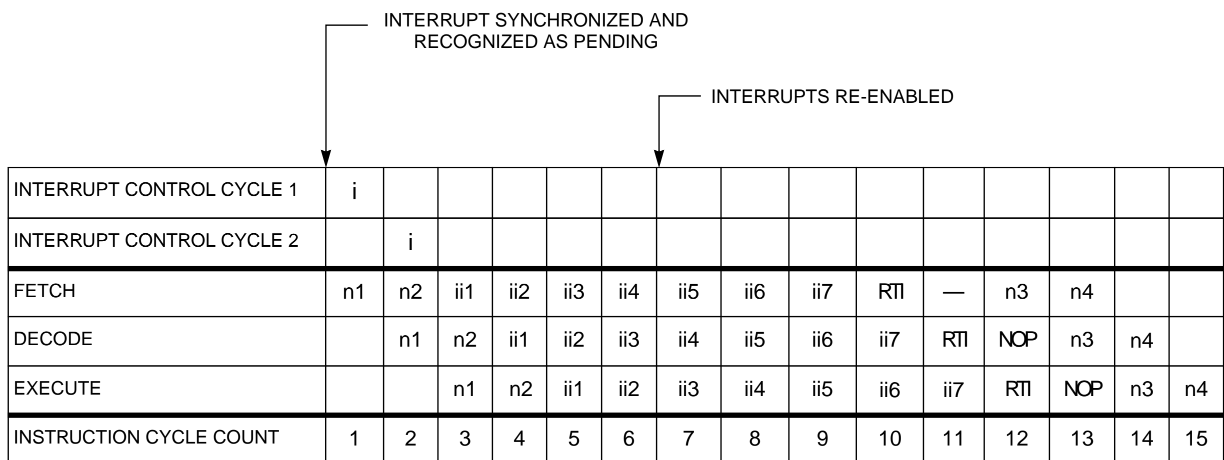
Figure 7-10 illustrates the effect of a long interrupt routine on the instruction pipeline. A short JSR (a JSR with 12-bit absolute address) is used to form the long interrupt routine. For this example, word 6 of the long interrupt routine is an RTI. The point at which interrupts are re-enabled and subsequent interrupts are allowed is shown to illustrate the non-interruptible nature of the early instructions in the long interrupt service routine.

Either one of the two instructions of the fast interrupt can be the JSR instruction that forms the long interrupt. Figure 7-11 and Figure 7-12 show the two possible cases. If the first fast interrupt vector instruction is the JSR, the second instruction is never used.

A REP instruction and the instruction that follows it are treated as a single two-word instruction, regardless of how many times it repeats the second instruction of the pair. Instruction fetches are suspended and will be reactivated only after the LC is decre-



(a) Instruction Fetches from Memory

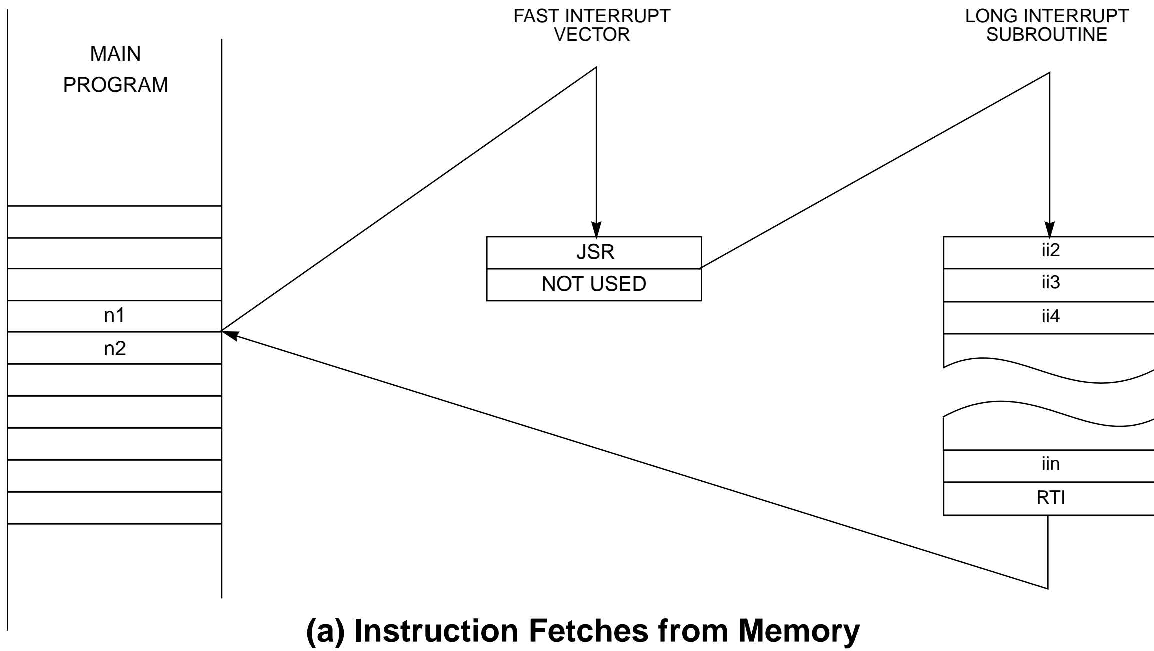


INTERRUPT CONTROL CYCLE 1	i														
INTERRUPT CONTROL CYCLE 2		i													
FETCH	n1	n2	ii1	ii2	ii3	ii4	ii5	ii6	ii7	RTI	—	n3	n4		
DECODE		n1	n2	ii1	ii2	ii3	ii4	ii5	ii6	ii7	RTI	NOP	n3	n4	
EXECUTE			n1	n2	ii1	ii2	ii3	ii4	ii5	ii6	ii7	RTI	NOP	n3	n4
INSTRUCTION CYCLE COUNT	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

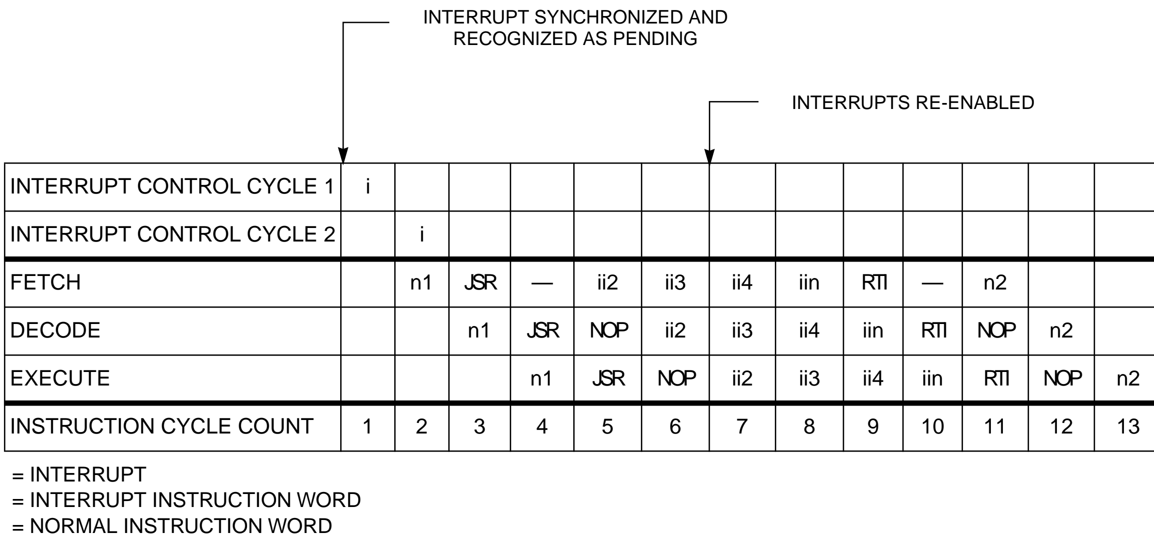
i = INTERRUPT
ii = INTERRUPT INSTRUCTION WORD
n = NORMAL INSTRUCTION WORD

(b) Program Controller Pipeline

Figure 7-10 Long Interrupt Service Routine



(a) Instruction Fetches from Memory

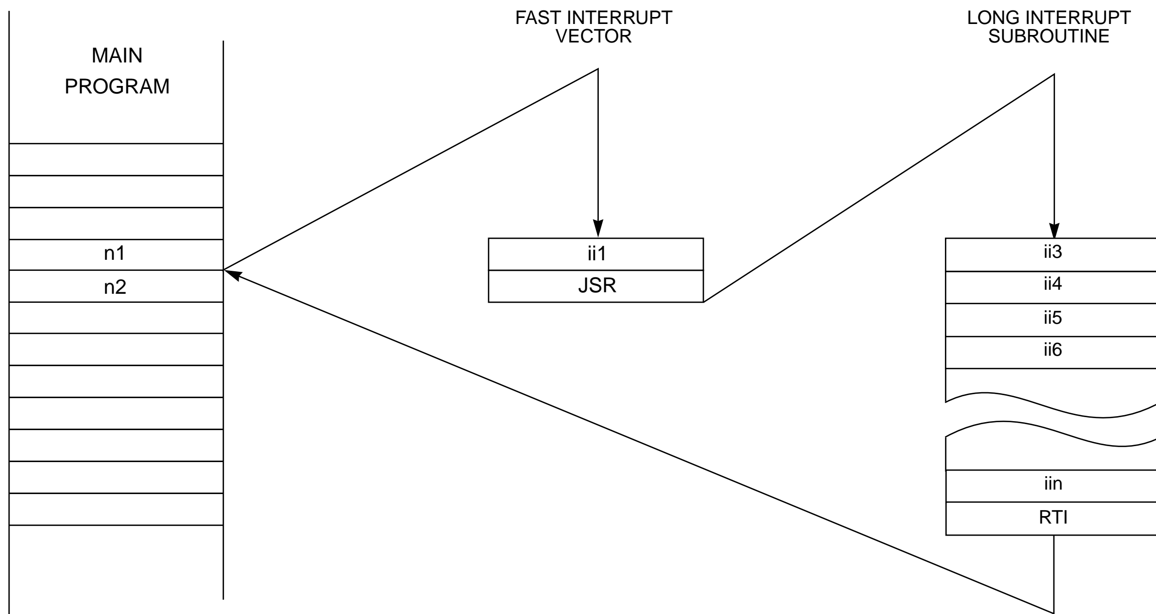


INTERRUPT CONTROL CYCLE 1	i												
INTERRUPT CONTROL CYCLE 2		i											
FETCH		n1	JSR	—	ii2	ii3	ii4	iin	RTI	—	n2		
DECODE			n1	JSR	NOP	ii2	ii3	ii4	iin	RTI	NOP	n2	
EXECUTE				n1	JSR	NOP	ii2	ii3	ii4	iin	RTI	NOP	n2
INSTRUCTION CYCLE COUNT	1	2	3	4	5	6	7	8	9	10	11	12	13

i = INTERRUPT
ii = INTERRUPT INSTRUCTION WORD
n = NORMAL INSTRUCTION WORD

(b) Program Controller Pipeline

Figure 7-11 JSR First Instruction of a Fast Interrupt



(a) Instruction Fetches from Memory

INTERRUPT SYNCHRONIZED AND RECOGNIZED AS PENDING

INTERRUPTS RE-ENABLED

INTERRUPT CONTROL CYCLE 1	i														
INTERRUPT CONTROL CYCLE 2		i													
FETCH		n1	ii1	JSR	—	ii3	ii4	ii5		iin	RTI	—	n2		
DECODE			n1	ii1	JSR	NOP	ii3	ii4	ii5	ii6	iin	RTI	NOP	n2	
EXECUTE				n1	ii1	JSR	NOP	ii3	ii4	ii5	ii6	iin	RTI	NOP	n2
INSTRUCTION CYCLE COUNT	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

i = INTERRUPT
ii = INTERRUPT INSTRUCTION WORD
n = NORMAL INSTRUCTION WORD

(b) Program Controller Pipeline

Figure 7-12 JSR Second Instruction of a Fast Interrupt

mented to one (see Figure 7-13). During the execution of n2 in Figure 7-13, no interrupts will be serviced. When LC finally decrements to one, the fetches are reinitiated, and pending interrupts can be serviced.

Sequential REP packages will cause pending interrupts to be rejected until the sequence of REP packages ends. REP packages are not interruptible because the instruction being repeated is not refetched. While that instruction is repeating, no instructions are fetched or decoded, and an interrupt can not be inserted. For example, in Figure 7-14, if n1, n3, and n5 are all REP instructions, no interrupts will be serviced until the last REP instruction (n5 and its repeated instruction, n6) completes execution.

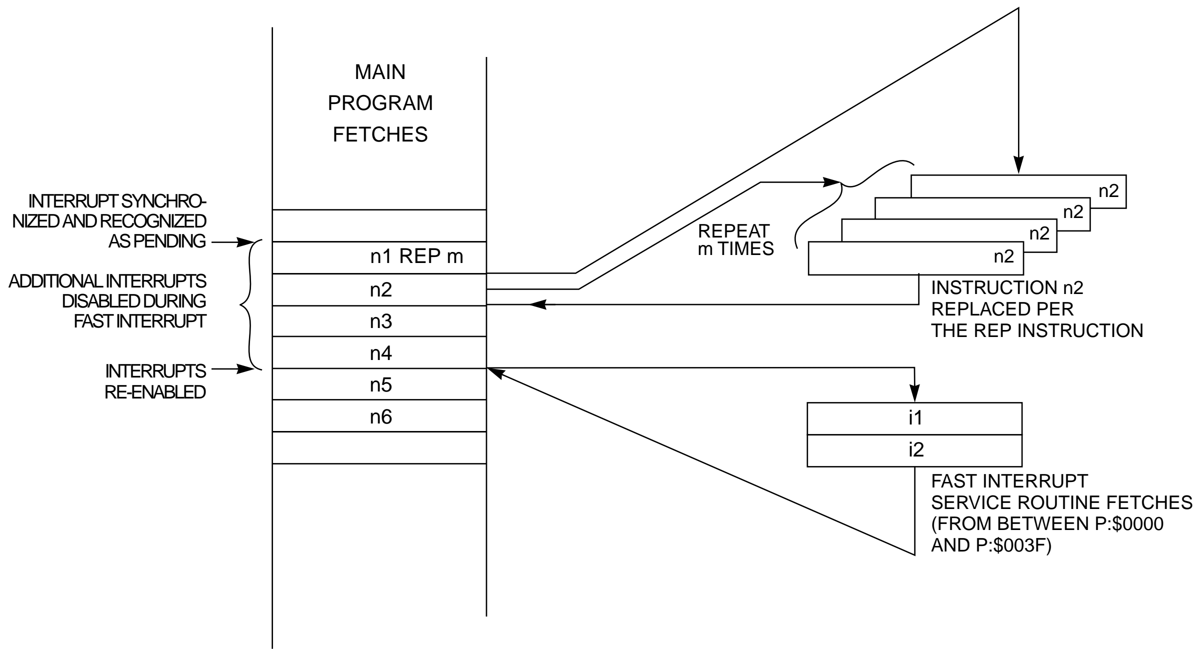
7.4 RESET PROCESSING STATE

The processor enters the reset processing state when a hardware reset occurs and the external RESET pin is asserted. The reset state:

1. resets internal peripheral devices;
2. sets the modifier registers to $FFFF;
3. clears the interrupt priority register;
4. sets the BCR to $FFFF, thereby inserting 15 wait states in all external memory accesses;
5. clears the stack pointer;
6. clears the scaling mode, trace mode, loop flag, double precision multiply mode, and condition code bits of the SR, and sets the interrupt mask bits of the SR;
7. clears the data ROM enable bit, the stop delay bit, and the internal Y memory disable bit, and
8. the DSP remains in the reset state until the RESET pin is deasserted.

When the processor deasserts the reset state:

9. it loads the chip operating mode bits of the OMR from the external mode select pins (MODA, MODB, MODC), and
10. begins program execution at program memory address defined by the state of bits MODA, MODB, and MODC in the OMR. The first instruction must be fetched and then decoded before executing. Therefore, the first instruction execution is two instruction cycles after the first instruction fetch.



i = INTERRUPT INSTRUCTION
n = NORMAL INSTRUCTION

(a) Instruction Fetches from Memory

INTERRUPT SYNCHRONIZED AND RECOGNIZED AS PENDING

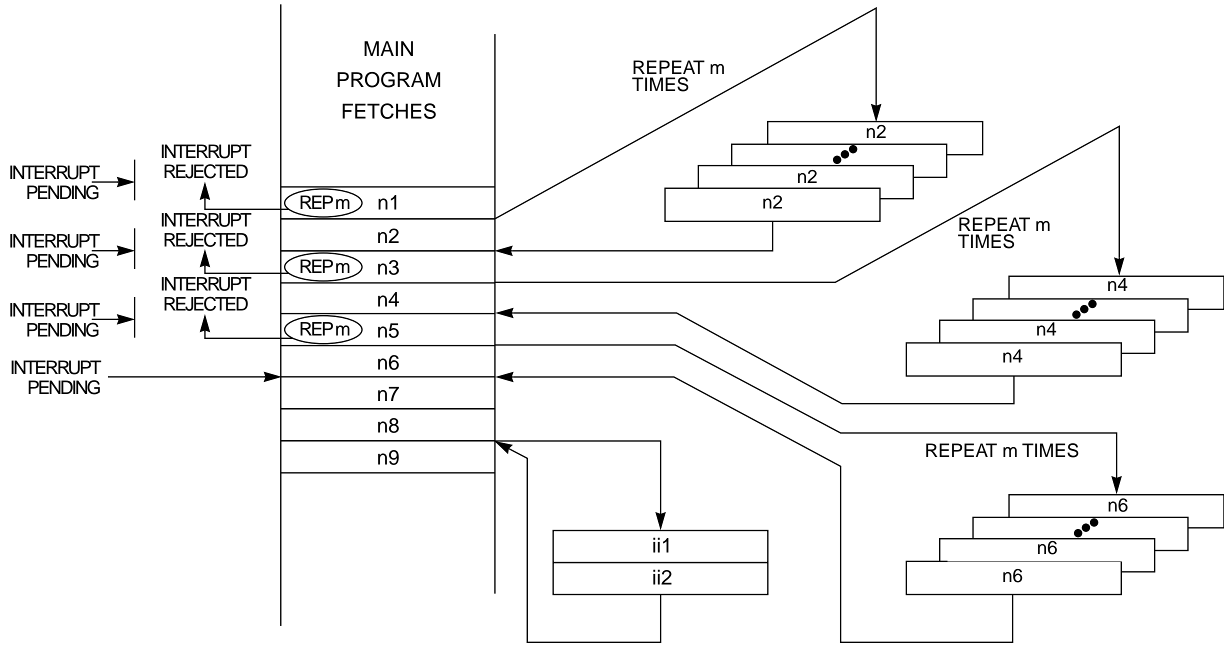
INTERRUPTS RE-ENABLED

INTERRUPT CONTROL CYCLE 1	i						i					
INTERRUPT CONTROL CYCLE 2		i%						i				
FETCH	REP	n2	n3					n4	ii1	ii2	n5	n6
DECODE		REP	NOP	n2	n2	n2	n2	n3	n4	ii1	ii2	n5
EXECUTE			REP	NOP	n2	n2	n2	n2	n3	n4	ii1	ii2
INSTRUCTION CYCLE COUNT	1	2	3	4	5	6	7	8	9	10	11	12

i = INTERRUPT
ii = INTERRUPT INSTRUCTION WORD
n = NORMAL INSTRUCTION WORD
i% = INTERRUPT REJECTED

(b) Program Controller Pipeline

Figure 7-13 Interrupting an REP Instruction



(a) Instruction Fetches from Memory

INTERRUPT SYNCHRONIZED AND RECOGNIZED AS PENDING

INTERRUPTS RE-ENABLED

INTERRUPT CONTROL CYCLE 1	i															i						
INTERRUPT CONTROL CYCLE 2		i%															i					
FETCH	REP	n2	REP				n4	REP				n6	n7				n8	ii1	ii2	n9		
DECODE		REP	NOP	n2	n2	n2	REP	NOP	n4	n4	n4	REP	NOP	n6	n6	n6	n7	n8	ii1	ii2	n9	
EXECUTE			REP	NOP	n2	n2	n2	REP	NOP	n4	n4	n4	REP	NOP	n6	n6	n6	n7	n8	ii1	ii2	n9
INSTRUCTION CYCLE COUNT	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22

i = INTERRUPT
ii = INTERRUPT INSTRUCTION WORD
n = NORMAL INSTRUCTION WORD
i% = INTERRUPT REJECTED

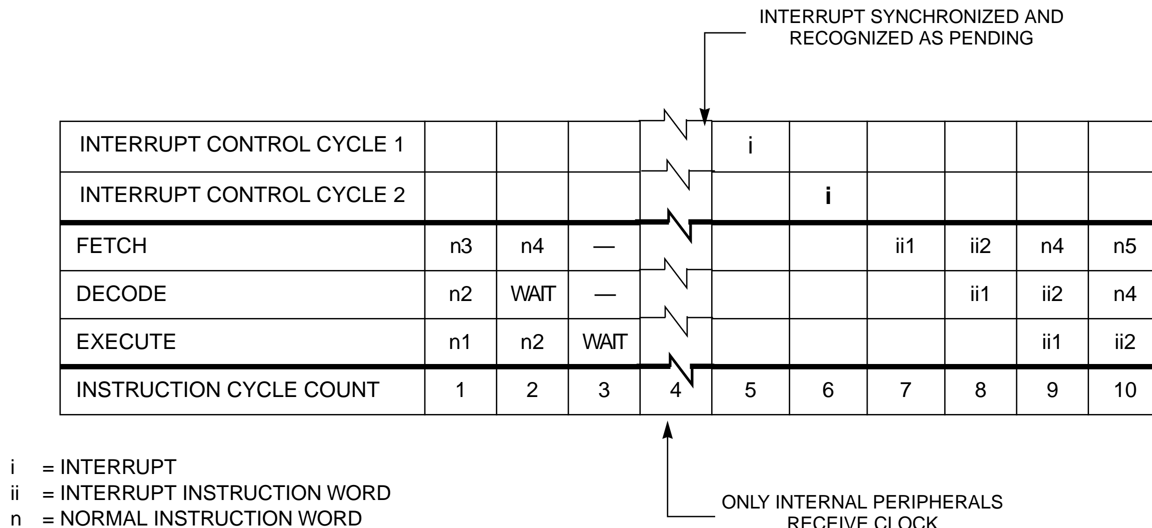
(b) Program Controller Pipeline

Figure 7-14 Interrupting Sequential REP Instructions

7.5 WAIT PROCESSING STATE

The WAIT instruction brings the processor into the wait processing state which is one of two low power-consumption states. Asserting the OnCE's debug request pin releases the DSP from the wait state. In the wait state, the internal clock is disabled from all internal circuitry except the internal peripherals. All internal processing is halted until an unmasked interrupt occurs, the Debug Request pin of the OnCE is asserted, or the DSP is reset.

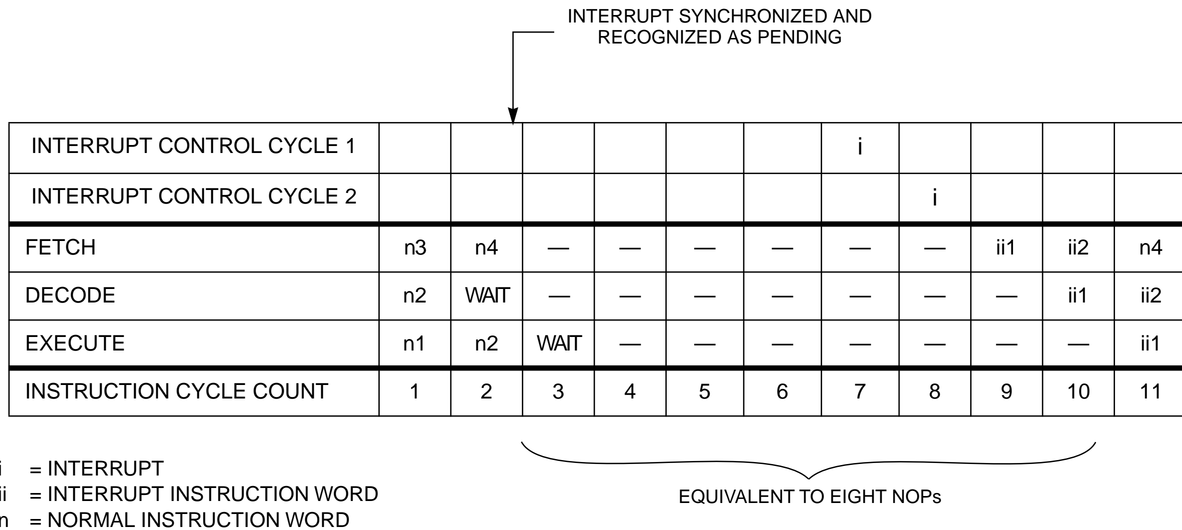
Figure 7-15 shows a WAIT instruction being fetched, decoded, and executed. It is fetched as n3 in this example and, during decode, is recognized as a WAIT instruction. The following instruction (n4) is aborted, and the internal clock is disabled from all internal circuitry except the internal peripherals. The processor stays in this state until an interrupt or reset is recognized. The response time is variable due to the timing of the interrupt with respect to the internal clock. Figure 7-15 shows the result of a fast interrupt bringing the processor out of the wait state. The two appropriate interrupt vectors are fetched and put in the instruction pipe. The next instruction fetched is n4, which had been aborted earlier. Instruction execution proceeds normally from this point.



INTERRUPT CONTROL CYCLE 1					i					
INTERRUPT CONTROL CYCLE 2						i				
FETCH	n3	n4	—				ii1	ii2	n4	n5
DECODE	n2	WAIT	—					ii1	ii2	n4
EXECUTE	n1	n2	WAIT						ii1	ii2
INSTRUCTION CYCLE COUNT	1	2	3	4	5	6	7	8	9	10

Figure 7-15 Wait Instruction Timing

Figure 7-16 shows an example of the WAIT instruction being executed at the same time that an interrupt is pending. Instruction n4 is aborted as before. The WAIT instruction causes a five-instruction-cycle delay from the time it is decoded, after which the interrupt is processed normally. The internal clocks are not turned off, and the net effect is that of executing eight NOP instructions between the execution of n2 and ii1.



INTERRUPT CONTROL CYCLE 1							i				
INTERRUPT CONTROL CYCLE 2								i			
FETCH	n3	n4	—	—	—	—	—	—	ii1	ii2	n4
DECODE	n2	WAIT	—	—	—	—	—	—	—	ii1	ii2
EXECUTE	n1	n2	WAIT	—	—	—	—	—	—	—	ii1
INSTRUCTION CYCLE COUNT	1	2	3	4	5	6	7	8	9	10	11

Figure 7-16 Simultaneous Wait Instruction and Interrupt

7.6 STOP PROCESSING STATE

The STOP instruction brings the processor into the stop processing state, which is the lowest power consumption state. In the stop state, the clock oscillator is gated off; whereas, in the wait state, the clock oscillator remains active. The chip clears all peripheral interrupts and external interrupts ($\overline{IRQA}$, $\overline{IRQB}$, and $\overline{NMI}$) when it enters the stop state. Trace or stack errors that were pending, remain pending. The priority levels of the peripherals remain as they were before the STOP instruction was executed. The on-chip peripherals are held in their respective individual reset states while in the stop state.

The stop processing state halts all activity in the processor until one of the following actions occurs:

1. A low level is applied to the $\overline{\text{IRQA}}$ pin.
2. A low level is applied to the $\overline{\text{RESET}}$ pin.
3. A low level is applied to the $\overline{\text{DR}}$ pin

Either of these actions will activate the oscillator, and, after a clock stabilization delay, clocks to the processor and peripherals will be re-enabled. The clock stabilization delay period is determined by the stop delay (SD) bit in the OMR.

The stop sequence is composed of eight instruction cycles called stop cycles. They are differentiated from normal instruction cycles because the fourth cycle is stretched for an indeterminate period of time while the four-phase clock is turned off.

The STOP instruction is fetched in stop cycle 1 of Figure 7-17, decoded in stop cycle 2 (which is where it is first recognized as a stop command), and executed in stop cycle 3. The next instruction (n4) is fetched during stop cycle 2 but is not decoded in stop cycle 3 because, by that time, the STOP instruction prevents the decode. The processor stops the clock and enters the stop mode. The processor will stay in the stop mode until it is restarted.

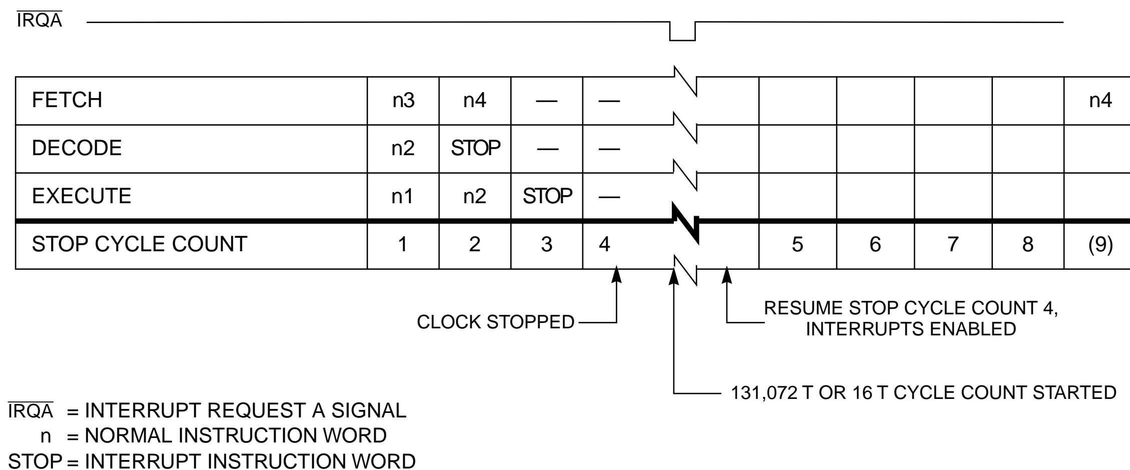


Figure 7-17 STOP Instruction Sequence

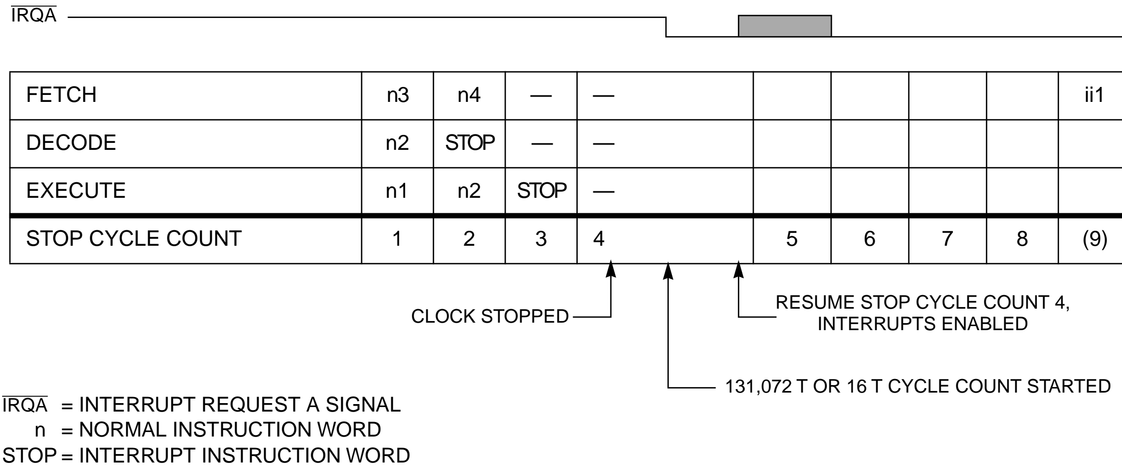


Figure 7-18 STOP Instruction Sequence Followed by $\overline{\text{IRQA}}$

Figure 7-18 shows the system being restarted by asserting the IRQA signal. If the exit from stop state was caused by a low level on the $\overline{\text{IRQA}}$ pin, then the processor will service the highest priority pending interrupt. If no interrupt is pending, then the processor resumes at the instruction following the STOP instruction that brought the processor into the stop state.

An $\overline{\text{IRQA}}$ deasserted before the end of the stop cycle count will not be recognized as pending. If $\overline{\text{IRQA}}$ is asserted when the stop cycle count completes, then an $\overline{\text{IRQA}}$ interrupt will be recognized as pending and will be arbitrated with any other interrupts.

Specifically, when $\overline{\text{IRQA}}$ is asserted, the internal clock generator is started and begins a delay determined by the SD bit of the OMR. When the chip uses the internal clock oscillator, the SD bit should be set to zero, to allow a longer delay time of 128K T cycles (131,072 T cycles) so that the clock oscillator may stabilize. When the chip uses a stable external clock, the SD bit may be set to one to allow a shorter (16 T cycle) delay time and a faster start up of the chip.

For example, assume that SD=0 so that the 128K T counter is used. During the 128K T count, the processor ignores interrupts until the last few counts and, at that time, begins to synchronize them. At the end of the 128K T cycle delay period, the chip restarts instruction processing, completes stop cycle 4 (interrupt arbitration occurs at this time), and executes stop cycles 5, 6, 7, and 8 (it takes 17T from the end of the 128K T delay to

the first instruction fetch). If the $\overline{\text{IRQA}}$ signal is released (pulled high) after a minimum of 4T but less than 128K T cycles, no $\overline{\text{IRQA}}$ interrupt will occur, and the instruction fetched after stop cycle 8 will be the next sequential instruction (n4 in Figure 7-18). An $\overline{\text{IRQA}}$ interrupt will be serviced as shown in Figure 7-18 if 1) the $\overline{\text{IRQA}}$ signal had previously been initialized as level sensitive, 2) $\overline{\text{IRQA}}$ is held low from the end of the 128K T cycle delay counter to the end of stop cycle count 8, and 3) no interrupt with a higher interrupt level is pending. If $\overline{\text{IRQA}}$ is not asserted during the last part of the STOP instruction sequence (6, 7, and 8) and if no interrupts are pending, the processor will refetch the next sequential instruction (n4). Since the $\overline{\text{IRQA}}$ signal is asserted (see Figure 7-18), the processor will recognize the interrupt and fetch and execute the instructions at P:$0008 and P:$0009 (the $\overline{\text{IRQA}}$ interrupt vector locations).

To ensure servicing $\overline{\text{IRQA}}$ immediately after leaving the stop state, the following steps must be taken before the execution of the STOP instruction:

1. Define $\overline{\text{IRQA}}$ as level sensitive – an edge-triggered interrupt will not be serviced.
2. Define $\overline{\text{IRQA}}$ priority as higher than the other sources and higher than the program priority.
3. Ensure that no stack error or trace interrupts are pending.
4. Execute the STOP instruction and enter the stop state.
5. Recover from the stop state by asserting the $\overline{\text{IRQA}}$ pin and holding it asserted for the whole clock recovery time. If it is low, the IRQA vector will be fetched. Also, the user must ensure that NMI will not be asserted during these last three cycles; otherwise, NMI will be serviced before $\overline{\text{IRQA}}$ because NMI priority is higher.
6. The exact elapsed time for clock recovery is unpredictable. The external device that asserts $\overline{\text{IRQA}}$ must wait for some positive feedback, such as specific memory access or a change in some predetermined I/O pin, before deasserting $\overline{\text{IRQA}}$.

The STOP sequence totals 131,104 T cycles (if SD=0) or 48 T cycles (if SD=1) in addition to the period with no clocks from the stop fetch to the $\overline{\text{IRQA}}$ vector fetch (or next instruction). However, there is an additional delay if the internal oscillator is used. An indeterminate period of time is needed for the oscillator to begin oscillating and then stabilize its amplitude. The processor will still count 131,072 T cycles (or 16 T cycles), but

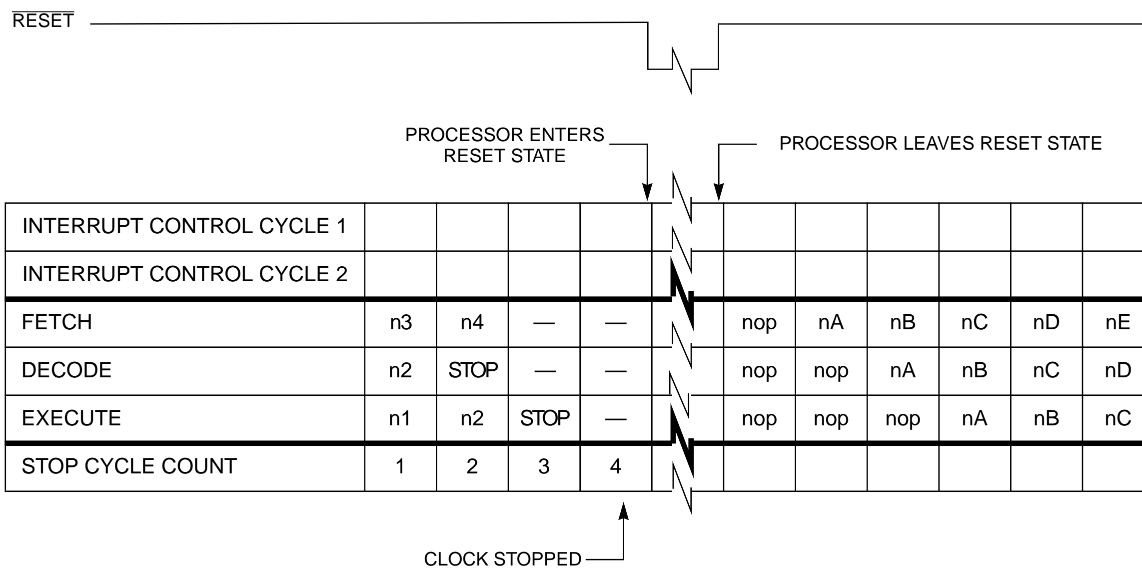
the period of the first oscillator cycles will be irregular; thus, an additional period of 19,000 T cycles should be allowed for oscillator irregularity (the specification recommends a total minimum period of 150,000 T cycles for oscillator stabilization). If an external oscillator is used that is already stabilized, no additional time is needed.

The PLL may be disabled or not when the chip enters the STOP state. If it is disabled and will not be re-enabled when the chip leaves the STOP state, the number of T cycles will be much greater because the PLL must regain lock.

If the STOP instruction is executed when the $\overline{\text{IRQA}}$ signal is asserted, the clock generator will not be stopped, but the four-phase clock will be disabled for the duration of the 128K T cycle (or 16 T cycle) delay count. In this case, the STOP looks like a 131,072 T + 35 T cycle (or 51 T cycle) NOP, since the STOP instruction itself is eight instruction cycles long (32 T) and synchronization of $\overline{\text{IRQA}}$ is 3T, which equals 35T.

A trace or stack error interrupt pending before entering the stop state is not cleared and will remain pending. During the clock stabilization delay, all peripheral and external interrupts are cleared and ignored (includes all SCI, SSI, HI, $\overline{\text{IRQA}}$, $\overline{\text{IRQB}}$, and NMI interrupts, but not trace or stack error). If the SCI, SSI, or HI have interrupts enabled in 1) their respective control registers and 2) in the interrupt priority register, then interrupts like SCI transmitter empty will be immediately pending after the clock recovery delay and will be serviced before continuing with the next instruction. If peripheral interrupts must be disabled, the user should disable them with either the control registers or the interrupt priority register before the STOP instruction is executed.

If $\overline{\text{RESET}}$ is used to restart the processor (see Figure 7-19), the 128K T cycle delay counter would not be used, all pending interrupts would be discarded, and the processor would immediately enter the reset processing state as described in Section 7.4. For example, the stabilization time recommended in theDSP56001 Technical Data Sheet for the clock ($\overline{\text{RESET}}$ should be asserted for this time) is only 50 T for a stabilized external clock but is the same 150,000 T for the internal oscillator. These stabilization times are recommended and are not imposed by internal timers or time delays. The DSP fetches instructions immediately after exiting reset. If the user wishes to use the 128K T (or 16 T) delay counter, it can be started by asserting $\overline{\text{IRQA}}$ for a short time (about two clock cycles).



INTERRUPT CONTROL CYCLE 1											
INTERRUPT CONTROL CYCLE 2											
FETCH	n3	n4	—	—		nop	nA	nB	nC	nD	nE
DECODE	n2	STOP	—	—		nop	nop	nA	nB	nC	nD
EXECUTE	n1	n2	STOP	—		nop	nop	nop	nA	nB	nC
STOP CYCLE COUNT	1	2	3	4							

$\overline{\text{IRESET}}$ = INTERRUPT
n = NORMAL INSTRUCTION WORD
nA, nB, nC = INSTRUCTIONS IN RESET ROUTINE
STOP = INTERRUPT INSTRUCTION WORD

Figure 7-19 STOP Instruction Sequence Recovering with $\overline{\text{RESET}}$