

# Chapter 2

## Core Architecture Overview

This chapter describes the DSP56300 family core, a powerful DSP engine that can execute an instruction on every clock cycle, yielding almost twice the performance of the Motorola DSP56000 core while retaining object code compatibility.

The DSP56300 core is composed of:

- External Memory Expansion Port (Port A)—See **Chapter 9**
- Data Arithmetic Logic Unit (Data ALU)—See **Chapter 3**
- Address Generation Unit (AGU)—See **Chapter 4**
- Instruction Cache Controller—See **Chapter 8**
- Program Control Unit (PCU)—See **Chapter 5**
- Direct Memory Access (DMA) Controller—See **Chapter 10**
- PLL Clock Generator—See **Chapter 6**
- JTAG Test Access Port and On-Chip Emulation (OnCE) module—See **Chapter 7**

To minimize the total system cost for customer applications, the DSP56300 core external memory interface, Port A, is powerful and versatile, providing a glueless interface to DRAMs (in some DSPs), SRAMs, and other memories via an on-chip DRAM controller (in some DSPs) as well as chip select logic. To assist with data movement over Port A and internally, the concurrent six-channel DMA augments the data throughput that characterizes DSP applications.

The core is designed for low power consumption in Normal and Wait and Stop modes. In Normal mode, only the blocks demanded for processing are active. Wait and Stop modes take the power savings a step further by closing down large portions of the core during periods of system inactivity. The integrated on-chip peripherals and memory (including instruction cache) also reduce power consumption by reducing the external bus accesses. As for the core execution units, only the memory modules being accessed consume power, so on-chip memory expansion does not increase power significantly. Limiting the external bus accesses saves on system power. Finally, the PLL can scale power consumption down with lower clock frequencies under user software control.

Low-power features of the DSP56300 family core include the following:

- Very low-power CMOS design
- Low-power Wait standby mode
- Ultra-low power Stop mode
- Power management units for further power reduction
- Fully static logic, with operation frequency down to DC

Sixteen-bit Compatibility mode enables full compatibility to object code written for the DSP56000 family of DSPs. Sixteen-bit Compatibility mode, which invokes 16-bit addressing capability, differs from the Sixteen-bit Arithmetic mode, which invokes 16-bit arithmetic operations. These modes are configured by two separate bits (SA and SC) in the Status Register (SR), which are described in **Chapter 5, Program Control Unit**.

## 2.1 Core Buses

The following 24-bit buses provide data exchange between the main core blocks:

Global Data Bus	GBD	Between Program Control Unit and other core structures
Peripheral I/O Expansion Bus	PIO_EB	To peripherals
Program Memory Expansion Bus	PM_EB	To Program ROM
Program Data Bus	PDB	Carries program data throughout the core
Program Address Bus	PAB	Carries program memory addresses throughout the core
X Memory Expansion Bus	XM_EB	To X memory
X Memory Data Bus	XDB	Carries X data throughout the core
X Memory Address Bus	XAB	Carries X memory addresses throughout the core
Y Memory Expansion Bus	YM_EB	To Y Memory
Y Memory Data Bus	YDB	Carries Y data throughout the core
Y Memory Address Bus	YAB	Carries Y memory addresses throughout the core
DMA Data Bus	DDB	Transfers data with DMA channels
DMA Address Bus	DAB	Transfers address information with DMA channels

**Figure 2-1** is a block diagram of the DSP56303, a member of the DSP56300 family. The diagram illustrates the core blocks of the DSP56300 family and shows representative peripherals for a DSP56300 family chip implementation.

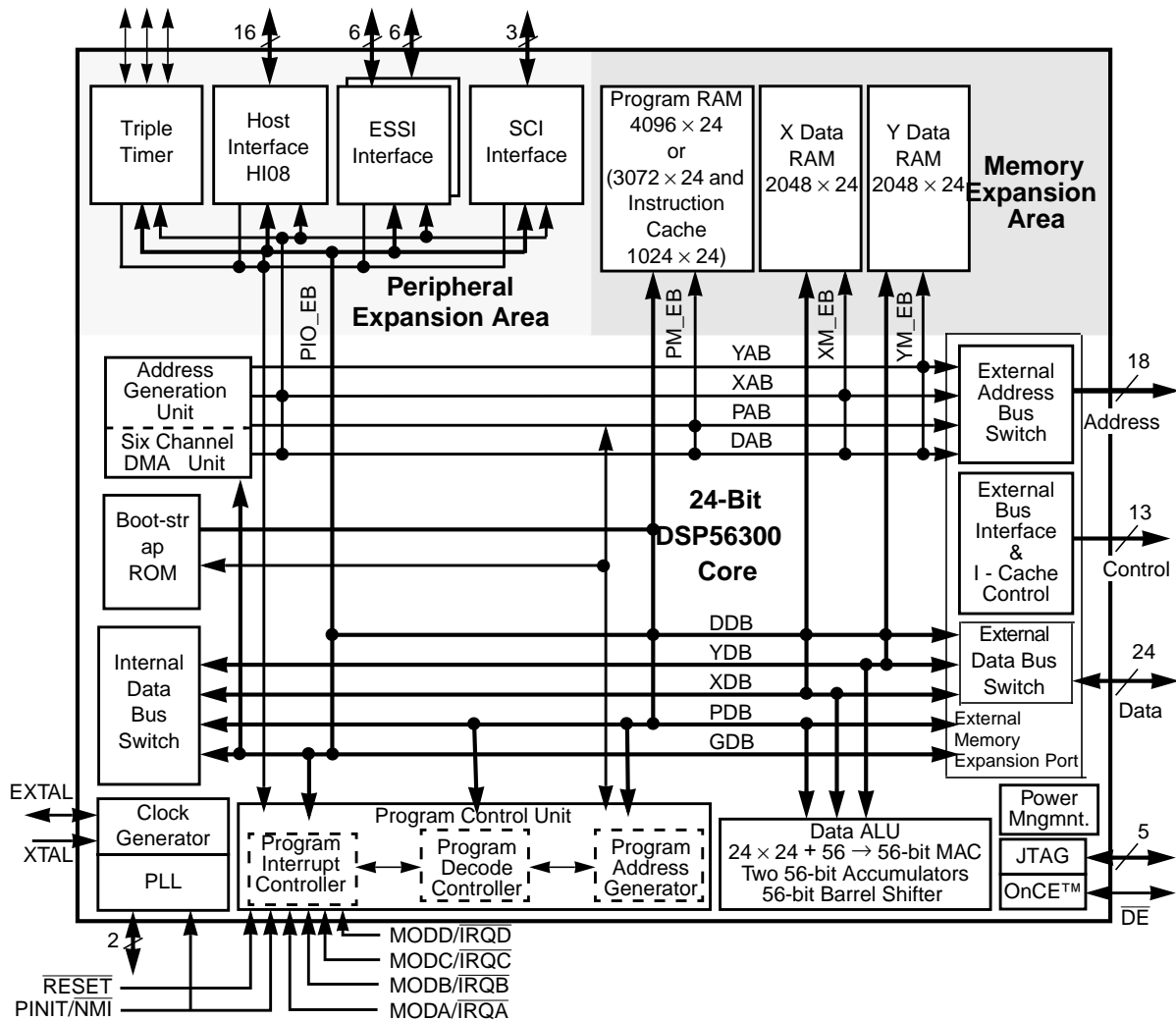


Figure 2-1. DSP56303 Block Diagram

**Note:** The registers in the core are discussed in detail in the chapters on the individual functional blocks.

## 2.2 Core Processing

As for all DSPs, the operation of the DSP56300 core is a combination of software and hardware interactions. This processing environment consists of the following components:

- **Instruction Set:** The instruction set provides the programming language for processing the algorithms required by specific applications. Appendix A contains a general overview of the instruction set and a description of the function and use of each instruction. Appendix B lists instruction execution timing and restrictions.

- *Core Modules:* These circuits transfer and modify data. They are generally configured through internal registers and activated or disabled by a combination of hardware signals (interrupts, request signals, etc.) and software. Chapters 3-10 of this document describe the structure and function of the various core modules.
- *Processing States:* Core processing states modify the operation of the core processor and the core modules that operate independently and in parallel to the core. These states include:
  - *Normal:* The typical operating mode in which code loads into the core processor and executes.
  - *Exception:* An event interrupts the normal execution flow. The processor halts normal processing and, depending on the event, may store the current operating environment, load a special handler program to respond to the exception, execute the handler program, and then return to normal execution flow. Typical exception causes can be software processing events or hardware service requests, such as peripheral or external device interrupts.
  - *Reset:* All execution halts and the processor and its registers in all peripherals are restored to a predetermined value that allows reloading of the executing code and reinitiation of the execution flow. Typically, if an operation has caused an unrecoverable error (that is, the handler cannot compensate for the exception event that halted normal processing), invoking the Reset mode, either by software or by asserting the physical  $\overline{\text{RESET}}$  signal, restores operational functioning.
  - *Wait:* Typically invoked by the WAIT instruction; the application requires only minimal processing. To save power, most operations stop until an event occurs that requires the processing to restart. Clock signals remain functional, so a quick restart is possible.
  - *Stop:* Typically invoked by using the STOP instruction; the application does not require immediate processing and a slow restart is acceptable (only if the PLL is disabled). All clock functions and operations halt, except for the ability to respond to an initiating event (that is,  $\overline{\text{RESET}}$ ,  $\overline{\text{DE}}$ , or  $\overline{\text{IRQA}}$ ).
  - *Debug:* Application developers can operate the system under the control of the JTAG Test Access Port and Boundary Scan function or the OnCE module. In this mode, an application can run a single instruction at a time, or sets of instructions at a time, until some defined event occurs, typically called a breakpoint.

## 2.3 Processing States

The following paragraphs describe the DSP56300 core processing states.

### 2.3.1 Normal Processing State

The Normal processing state is associated with instruction execution. DSP56300 core instructions execute in a seven-stage pipeline, typically at a rate of one instruction every clock cycle. However, the following instructions require additional time to execute:

- All double-word instructions
- Instructions with an addressing mode that requires more than one cycle for the address calculation
- Instructions causing a change of flow

Instruction pipelining allows overlapping of instruction execution so that a pipeline stage of a given instruction occurs concurrently with pipeline stages of other instructions. Only one word is fetched per cycle, so for double-word instructions, the second word of an instruction is fetched before the next instruction is fetched. **Table 2-1** describes the seven stages of the DSP56300 core pipeline. n1 and n2 in **Table 2-1** refer to first and second instructions, respectively. The third instruction, n3, which contains an instruction extension word, n3e, takes two clock cycles to execute. The extension word is either an absolute address or immediate data. Although it takes seven clock cycles for the pipeline to fill and the first instruction to execute, a further instruction usually completes on each clock cycle.

Each instruction requires a minimum of seven clock cycles to fetch, decode, and execute. This results in a delay of seven clock cycles from power-up to fill the pipeline. A new instruction may begin immediately following the previous instruction. Two-word instructions require a minimum of eight clock cycles to execute (seven cycles for the first instruction word to move through the pipe and execute and one more cycle for the second word to execute). For a complete description of the execution timing of the various instructions, see **Appendix A, Instruction Timing and Restrictions**.

**Table 2-1** Instruction Pipeline

Operation	Instruction Cycle										
	1	2	3	4	5	6	7	8	9	10	11
Fetch 1	n1	n2	n3	n3e	n4	n5	n6	n7	n8	n9	n10
Fetch 2		n1	n2	n3	n3e	n4	n5	n6	n7	n8	n9
Decode			n1	n2	n3	n3e	n4	n5	n6	n7	n8

**Table 2-1** Instruction Pipeline

Operation	Instruction Cycle										
	1	2	3	4	5	6	7	8	9	10	11
Address Gen 1				n1	n2	n3	n3e	n4	n5	n6	n7
Address Gen 2					n1	n2	n3	n3e	n4	n5	n6
Execute 1						n1	n2	n3	n3e	n4	n5
Execute 2							n1	n2	n3	n3e	n4

### 2.3.2 Exception Processing State (Interrupt Processing)

The Exception Processing state is associated with interrupts that are generated by conditions inside the DSP or by external sources. There are many sources for interrupts to the DSP56300 core, some generating more than one interrupt. An interrupt vector scheme with 128 vectors of defined priority provides fast interrupt service. Interrupt processing in the DSP56300 core proceeds as follows:

1. A hardware interrupt is synchronized with the DSP56300 core 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 the interrupt to be processed. The arbiter automatically ignores any interrupts with an Interrupt Priority Level (IPL) lower than the interrupt mask level in the SR and selects the remaining interrupt with the highest IPL.
3. The interrupt controller freezes the program counter (PC) and fetches two instructions at the two interrupt vector addresses associated with the selected interrupt.
4. The interrupt controller inserts 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.

When a fast interrupt executes, the state of the machine is not saved on the stack if neither of the two instructions is a Jump To Subroutine (JSR) instruction (for example, a JSCLR). A long interrupt executes 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 from where the next instruction is fetched.

**Note:** Any Jump to Subroutine (JSR) instruction makes the interrupt long (for example, JScc, BSSET, etc.).

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 minimum overhead is often desirable. This limited context switch is accomplished by a fast interrupt. The long interrupt is used when a more complex task must be accomplished to service the interrupt.

Exceptions may originate from any of the 128 vector addresses listed in **Table 2-2**.

Exceptions may originate from one of two groups: core and peripherals. **Table 2-2** lists only the sources originating from the core. For sources originating from peripherals, see the device-specific user's manual. **Table 2-2** shows the corresponding interrupt starting address for each interrupt source. These addresses reside in the 256 locations of program memory to which the Vector Base Address Register VBA in the PCU points. When an interrupt is serviced, 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 DSP56300 core is said to be vectored. A vectored interrupt structure has low overhead execution. If certain interrupts will definitely not be used, their vector locations can be used for program or data storage.

**Table 2-2** Interrupt Sources

Interrupt Starting Address	Interrupt Priority Level (IPL)	Interrupt Source
VBA:\$00	3	Hardware RESET
VBA:\$02	3	Stack Error
VBA:\$04	3	Illegal Instruction
VBA:\$06	3	Debug Request Interrupt
VBA:\$08	3	Trap
VBA:\$0A	3	Non-Maskable Interrupt ( $\overline{\text{NMI}}$ )
VBA:\$0C	3	Reserved for Future Level—3 Interrupt Source
VBA:\$0E	3	Reserved for Future Level—3 Interrupt Source
VBA:\$10	0–2	$\overline{\text{IRQA}}$
VBA:\$12	0–2	$\overline{\text{IRQB}}$
VBA:\$14	0–2	$\overline{\text{IRQC}}$
VBA:\$16	0–2	$\overline{\text{IRQD}}$
VBA:\$18	0–2	DMA Channel 0
VBA:\$1A	0–2	DMA Channel 1
VBA:\$1C	0–2	DMA Channel 2
VBA:\$1E	0–2	DMA Channel 3
VBA:\$20	0–2	DMA Channel 4

**Table 2-2** Interrupt Sources (Continued)

Interrupt Starting Address	Interrupt Priority Level (IPL)	Interrupt Source
VBA:\$22	0–2	DMA Channel 5
VBA:\$24	0–2	Peripheral interrupt request 1
VBA:\$26	0–2	Peripheral interrupt request 2
:	:	
VBA:\$FE	0–2	Peripheral interrupt request 110

The 128 interrupts are prioritized into four levels. Level 3, the highest priority level, is not maskable. Levels 0 – 2 are maskable. The interrupts within each level are prioritized.

### 2.3.2.1 Hardware Interrupt Source

Two types of hardware interrupts to the DSP56300 core exist: internal and external. The internal interrupts come from on-chip sources:

- Stack Error
- Illegal Instruction
- Debug Request
- Trap
- DMAs
- Peripherals

Each internal interrupt source is serviced if it is not masked. When serviced, the interrupt request is cleared. Each maskable, internal interrupt source has independent enable control. The external hardware interrupts are:  $\overline{\text{NMI}}$ ,  $\overline{\text{IRQA}}$ ,  $\overline{\text{IRQB}}$ ,  $\overline{\text{IRQC}}$ , and  $\overline{\text{IRQD}}$ . The  $\overline{\text{NMI}}$  interrupt is an edge-triggered, Non-Maskable Interrupt (NMI) for use in software development, watch-dog, power fail detect, etc. The  $\overline{\text{IRQA}}$ ,  $\overline{\text{IRQB}}$ ,  $\overline{\text{IRQC}}$  and  $\overline{\text{IRQD}}$  interrupts can be programmed to be level-sensitive or edge-triggered. Since the level-sensitive interrupts are not automatically cleared when they are serviced, they must be cleared by other means before the end of the interrupt routine because multiple interrupts must be prevented. Usually, external hardware detects the interrupt acknowledge of the core interrupt and removes the interrupt request source.

The edge-triggered interrupts are latched as pending on the high-to-low transition of the interrupt input and are automatically cleared when the interrupt is serviced.  $\overline{\text{IRQA}}$ ,  $\overline{\text{IRQB}}$ ,

$\overline{IRQC}$  and  $\overline{IRQD}$  can be programmed to one of three priority levels: 0, 1, or 2, all of which are maskable. Additionally, these interrupts have independent enable control.

When the  $\overline{IRQA}$ ,  $\overline{IRQB}$ ,  $\overline{IRQC}$  and  $\overline{IRQD}$  interrupts are disabled in the interrupt priority register, the pending request is ignored, regardless of whether the interrupt input was defined as level-sensitive or edge-triggered. Additionally, as long as an interrupt (edge or level sensitive) is disabled, its detection latch remains in the Reset state. If the level-sensitive interrupt is disabled while the interrupt is pending, the pending interrupt is cancelled. However, if the interrupt has been fetched, it is not cancelled.

**Note:** On all external, level-sensitive interrupt sources, the interrupt should be serviced (that is, the interrupt source cleared) by the instructions at the interrupt vector for a fast interrupt, or by a long interrupt routine.

### 2.3.2.2 Software Interrupt Sources

There are two software interrupt sources:

- *Illegal Instruction Interrupt (III)*: The III is a Non-Maskable Interrupt (IPL 3) that is serviced immediately after the illegal instruction executes or attempts to execute (any undefined operation code).
- *TRAP*: A Non-Maskable Interrupt (IPL 3) that is serviced immediately after the TRAP or TRAPcc instruction executes (condition true).

### 2.3.2.3 Interrupt Priority Structure

Four interrupt priority levels (IPLs) exist. IPLs are numbered from 0 (the lowest level) to 3 (the highest level). IPLs 0, 1, and 2 are maskable. Level 3 is non-maskable. The IPL 3 interrupts are:

- Hardware Reset
- Illegal Instruction Interrupt (III)
- Stack Error
- TRAP
- NMI
- Debug

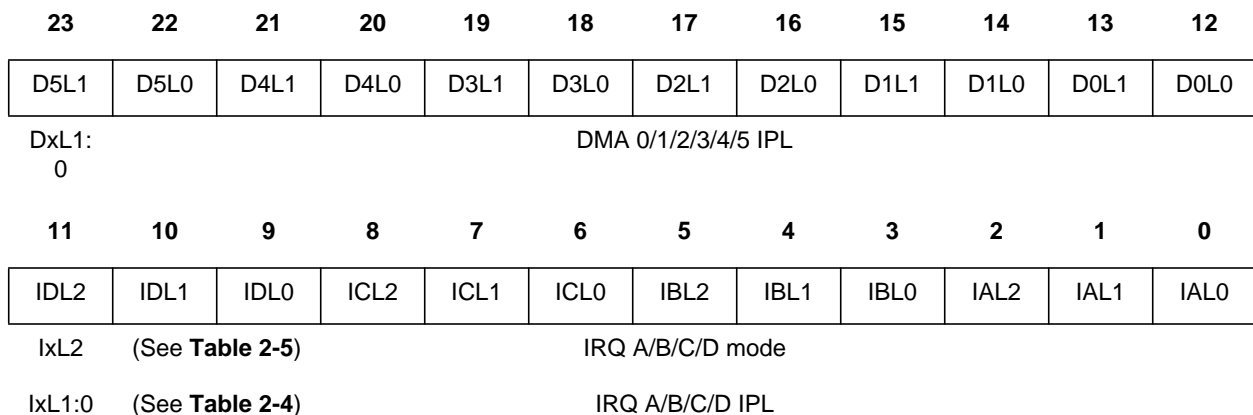
The interrupt mask bits (I1, I0) in the SR reflect the current processor priority level and indicate the IPL needed for an interrupt source to interrupt the processor (see **Table 2-3**). Interrupts are inhibited for all priority levels less than the current processor priority level. However, level 3 interrupts are not maskable and therefore can always interrupt the processor.

**Table 2-3** Status Register Interrupt Mask Bits

I1	I0	Interrupts Permitted	Interrupts 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

For details on the Status Register, see **Chapter 5, Program Control Unit**.

The DSP56300 core has two interrupt priority registers: IPRC that is dedicated for DSP56300 core interrupt sources and IPRP that is dedicated for the peripheral interrupt sources specific to the chip. These control registers are mapped on the internal X I/O memory space. The Interrupt Priority Level (IPL) for each interrupt source is software programmable. Each on-chip or external peripheral device can be programmed to one of the three maskable priority levels (IPL 0, 1, or 2). IPLs are set by writing to the interrupt priority registers shown in **Figure 2-1** and **Figure 2-2**. These two read/write registers specify the IPL for each of the interrupting devices. In addition, the IPRC register specifies the trigger mode of each external interrupt source and enables or disables the individual external interrupts. These registers are cleared on hardware reset or by the RESET instruction. **Table 2-4** defines the IPL bits. **Table 2-5** defines the External Interrupt Trigger mode bit.

**Figure 2-1.** Interrupt Priority Register C (IPRC)

23	22	21	20	19	18	17	16	15	14	13	12
PerCL 1	PerCL 0	PerBL 1	PerBL 0	PerAL 1	PerAL 0	Per9L 1	Per9L 0	Per8L 1	Per8L 0	Per7L 1	Per7L 0
11	10	9	8	7	6	5	4	3	2	1	0
Per6L 1	Per6L 0	Per5L 1	Per5L 0	Per4L 1	Per4L 0	Per3L 1	Per3L 0	Per2L 1	Per2L 0	Per1L 1	Per1L 0

**Figure 2-2.** Interrupt Priority Register P (IPRP)**Table 2-4** Interrupt Priority Level Bits

IxL1	IxL0	Enabled	IPL
0	0	No	—
0	1	Yes	0
1	0	Yes	1
1	1	Yes	2

**Table 2-5** External Interrupt Trigger Mode Bit

IxL2	Trigger Mode
0	Level
1	Negative Edge

If more than one exception is pending when an instruction executes, the interrupt with the highest priority level is serviced first. When multiple interrupt requests with the same IPL are pending, a second fixed-priority structure within that IPL determines which interrupt is serviced. **Table 2-6** shows the interrupt priority for all interrupts.

**Table 2-6** Exception Priorities Within an IPL

Priority	Exception
<b>Level 3 (Nonmaskable)</b>	
<b>Highest</b>	Stack Error
	Illegal Instruction
	Debug Request Interrupt

**Table 2-6** Exception Priorities Within an IPL (Continued)

Priority	Exception
	Trap
	Non-Maskable Interrupt (NMI)
<b>Lowest</b>	Non-Maskable Peripheral Interrupt
<b>Levels 0, 1, 2 (Maskable)</b>	
<b>Highest</b>	$\overline{\text{IRQA}}$ (External Interrupt)
	$\overline{\text{IRQB}}$ (External Interrupt)
	$\overline{\text{IRQC}}$ (External Interrupt)
	$\overline{\text{IRQD}}$ (External Interrupt)
	DMA Channel 0 Interrupt
	DMA Channel 1 Interrupt
	DMA Channel 2 Interrupt
	DMA Channel 3 Interrupt
	DMA Channel 4 Interrupt
	DMA Channel 5 Interrupt
<b>Lowest</b>	Peripheral interrupt sources*
*See device-specific user's manual NOTE: The higher-priority interrupt is at the lower vector address.	

### 2.3.2.4 Instructions Preceding the Interrupt Instruction Fetch

The following conditions apply to instructions preceding an interrupt instruction fetch:

- Every instruction requiring more than one cycle to execute is aborted when it is fetched in the cycle preceding the fetch of the first interrupt instruction word.
- Aborted instructions are fetched again 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-one-cycle instruction, that instruction completes normally before the start of the interrupt routine.
- During an interrupt instruction fetch, two instruction words are fetched — the first from the interrupt starting address and the second from the next address.

### 2.3.2.5 Interrupt Types

Two types of interrupt routines can be used: fast and long. The fast routine consists of the two automatically inserted interrupt instruction words. These words can be any unrestricted, single two-word instruction or any two unrestricted one-word instructions, except RTI or RTS. Fast interrupt routines are not interruptible.

**Note:** Status is not preserved during a fast interrupt routine; therefore, instructions that modify status should not be used at the interrupt starting address or next address.

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 next address:

1. The PC (containing the return address) and the SR are stacked.
2. The Loop flag is cleared.
3. The Scaling mode bits (S[1 – 0]) in the Status Register (SR) are cleared.
4. The Sixteen-bit Arithmetic (SA) mode bit is cleared.
5. The IPL is raised to disallow further interrupts of the same or lower levels. See **Table 2-6**.

Only the long interrupt routine should be terminated by an RTI. Long interrupt routines are interruptible by higher-priority interrupts.

**Note:** Do not use RTI for fast interrupts.

### 2.3.2.6 Interrupt Arbitration

External interrupts are internally synchronized with the processor clock before their interrupt-pending flags are set. Each external interrupt and internal interrupt has its own flag. After each instruction executes, all interrupts are arbitrated (that is, all hardware interrupts that have been latched into their respective interrupt-pending flags and all internal 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 priority shown in **Table 2-6**, 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. The interrupt-pending flag for the chosen interrupt is not cleared until the second interrupt vector of the chosen interrupt is fetched. A new interrupt from the same source is not accepted for the next interrupt arbitration until the interrupt-pending flag is cleared..

### 2.3.2.7 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 again for the subsequent address after that. While the interrupt instructions are being fetched, the PC is not updated. After the two interrupt words have been fetched, the PC is used for any subsequent instruction fetches.

### 2.3.2.8 Interrupt Instruction Execution

Interrupt instruction execution is considered “fast” if neither of the instructions of the interrupt service routine cause 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 that normally contains only a JMP instruction at the exception start address. Almost any instruction can be used in a fast interrupt routine. A fast interrupt routine may contain either two single-word instructions or one double-word instruction. **Table 2-7** shows the effect of a fast interrupt routine on the instruction pipeline. The fast interrupt executes only two instructions (ii1 and ii2) and then automatically resumes execution of the main program. **Table 2-8** shows the effect of a long interrupt routine on the instruction pipeline. A short JSR (ii1) is used to call the long interrupt routine which includes the four instructions sr1, sr2, sr3 and an rti. Instructions ii2, n3, sr5 and sr6 are neither decoded nor executed.

**Table 2-7** Fast Interrupt Pipeline

Operation	Instruction Cycle											
	1	2	3	4	5	6	7	8	9	10	11	12
Fetch 1	n1	n2	ii1	ii2	n3	n4						
Fetch 2		n1	n2	ii1	ii2	n3	n4					
Decode			n1	n2	ii1	ii2	n3	n4				
Address Gen 1				n1	n2	ii1	ii2	n3	n4			
Address Gen 2					n1	n2	ii1	ii2	n3	n4		

**Table 2-7** Fast Interrupt Pipeline

Operation	Instruction Cycle											
	1	2	3	4	5	6	7	8	9	10	11	12
Execute 1						n1	n2	ii1	ii2	n3	n4	
Execute 2							n1	n2	ii1	ii2	n3	n4
n = normal instruction word ii = interrupt instruction word												

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

1. The processor status is not saved.
2. The fast interrupt routine can modify the status of the normal instruction stream (for example, use the DO instruction, but such instructions should not be used in order to assure proper operation).
3. The PC, which contains the address of the next instruction to be executed in normal processing, remains unchanged during a fast interrupt routine.
4. The fast interrupt returns without an RTI.
5. Normal instruction fetching resumes using the PC following the completion of the fast interrupt routine.
6. A fast interrupt is not interruptible.
7. A JSR instruction within the fast interrupt routine forms a long interrupt routine.

**Table 2-8** Long Interrupt Pipeline

Operation	Instruction Cycle															
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Fetch 1	n1	n2	ii1	ii2	n3	sr1	sr2	sr3	sr4	sr5	sr6	n3	n4	n5	n6	n7
Fetch 2		n1	n2	jsr	ii2	n3	sr1	sr2	sr3	rti	sr5	sr6	n3	n4	n5	n6
Decode			n1	n2	jsr	—	—	sr1	sr2	sr3	rti	—	—	n3	n4	n5
Addr. Gen 1				n1	n2	jsr	—	—	sr1	sr2	sr3	rti	—	—	n3	n4
Addr. Gen 2					n1	n2	jsr	—	—	sr1	sr2	sr3	rti	—	—	n3
Execute 1						n1	n2	jsr	—	—	sr1	sr2	sr3	rti	—	—
Execute 2							n1	n2	jsr	—	—	sr1	sr2	sr3	rti	—
n = normal instruction word ii = interrupt instruction word sr = service routine word																

Execution of a long interrupt routine always adheres 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 and Scaling mode bits in the Status Register are cleared.
3. The interrupt service routine can be interrupted (that is, nested interrupts are supported), but can only be interrupted by a higher priority interrupt.
4. The long interrupt routine, which can be any length, should be terminated by an RTI, which restores the PC and SR from the stack.

Either of the two instructions of the fast interrupt can be the JSR instruction that forms the long interrupt.

**Note:** A REP instruction is 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 decremented to one. During the execution of the repeated instruction, no interrupts are serviced. When LC finally decrements to one, the fetches are reinitiated, and pending interrupts are serviced.

If a non-interruptible code sequence is desired, change the IPL bits to the desired mask level. Due to pipelining, you will need four instructions before you can guarantee that the code is not interrupted by a maskable interrupt.

### 2.3.3 Reset Processing State

The DSP device enters reset processing state when the external  $\overline{\text{RESET}}$  pin is asserted (a hardware reset). In the Reset state:

1. Internal peripheral devices are reset.
2. The modifier registers (M0–M7) are set to \$FFFFFF.
3. The interrupt priority registers are cleared.
4. The Bus Control Register (BCR), the Address Attribute Registers (AAR3–AAR0) and the DRAM Control Register (DCR) are set to their initial values as described in **Chapter 9, External Memory Interface (Port A)**. The initial value causes a maximum number of wait states to be added to every external memory access.
5. The Stack Pointer (SP) and the Stack Counter (SC) are cleared.
6. The following bits of the SR are cleared:

- Rounding mode (RM) bit (Bit 21)
  - Arithmetic Saturation mode (SM) bit (Bit 20)
  - Cache Enable (CE) bit (Bit 19)
  - Sixteen-bit Arithmetic (SA) mode bit (Bit 17)
  - DO Forever (FV) flag bit (Bit 16)
  - DO Loop Flag (LF) bit (Bit 15)
  - Double Precision Multiply (DM) mode bit (Bit 14)
  - Sixteen-bit Compatibility (SC) mode bit (Bit 13)
  - Scaling (S[1 – 0]) bits (Bit 11 and Bit 10)
  - Condition Code bits (SR[7 – 0])
7. The following bits of the SR are set:
    - Core Priority (CP[1 – 0]) bits (Bit 23 and Bit 22)
    - Interrupt (I[1 – 0]) mask bits (Bit 9 and Bit 8)
  8. The Instruction Cache Controller is initialized as described in **Chapter 8**, *Instruction Cache*.
  9. The Cache Enable (CE) bit in SR and the Burst mode bit in OMR are cleared.
  10. The PLL Control register is initialized as described in **Chapter 6**, *PLL and Clock Generator*.
  11. The Vector Base Address Register (VBA) is cleared.

The DSP56300 core remains in the Reset state until  $\overline{\text{RESET}}$  is deasserted. Upon leaving the Reset state, the Chip Operating mode bits of the OMR are loaded from the external mode select pins (MODA, MODB, MODC, MODD), and program execution begins at the program memory address as described in **Chapter 11**, *Operating Modes and Memory Spaces*.

### 2.3.4 Wait Processing State

The Wait processing state is a low-power consumption state that occurs when the WAIT instruction executes. In the Wait state, the internal clock is disabled from all internal circuitry except the internal peripherals. All internal processing halts until an unmasked interrupt occurs, the DSP is reset, or  $\overline{\text{DE}}$  is asserted. If the exit from Wait state is caused by asserting  $\overline{\text{DE}}$ , the processor enters the Debug mode.

### 2.3.5 Stop Processing State

The Stop processing state is the lowest power consumption mode that occurs when the STOP instruction executes. In Stop mode, the clock oscillator activity depends on the PSTP bit in the PLL control register. If this bit is cleared, the clock oscillator is turned off. If the bit is set, the VCO remains active and the global clock to the entire chip is disabled. All activity in the processor halts until one of the following actions occurs:

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

Any of these actions enables the oscillator and, after a clock stabilization delay, clocks to the processor and peripherals are re-enabled. When the clocks to the processor and peripherals are re-enabled:

1. If the exit from Stop state was caused by a low level on the  $\overline{\text{RESET}}$  pin, then the processor enters the Reset processing state.
2. If the exit from Stop state was caused by a low level on the  $\overline{\text{IRQA}}$  pin, then the processor services the highest-priority pending interrupt. If no interrupt is pending (i. e.  $\overline{\text{IRQA}}$  was negated before interrupts were arbitrated), or if no interrupt is enabled, the processor resumes execution at the instruction following the STOP instruction that caused the entry into the Stop state.
3. If the exit from Stop state was caused by a low level on the  $\overline{\text{DE}}$  pin, then the processor enters the Debug mode.

For minimum power consumption during the Stop state at the cost of longer recovery time, clear the PSTP bit of the PLL Control Register. To enable rapid recovery when exiting the Stop state, at the cost of higher power consumption, set PSTP. PSTP is cleared by hardware reset.

### 2.3.6 Debug State

Debug state is invoked and used with the JTAG/OnCE port. See **Chapter 7, Debugging Support** for a description of the Debug state.