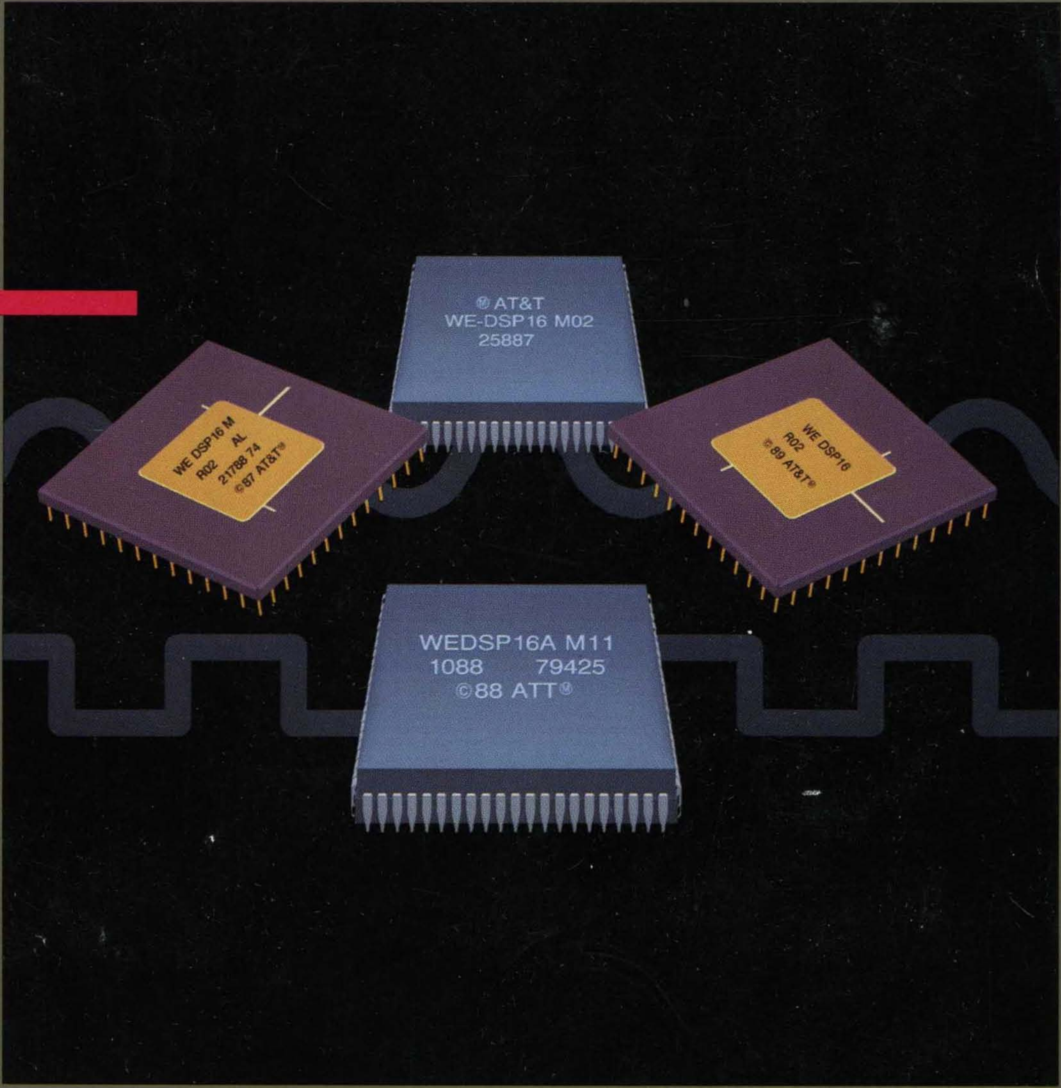
Information Manual

AT&T

WE® DSP16 and DSP16A Digital Signal Processor



AT&T

January 1989

WE ® DSP16 and DSP16A Digital Signal Processors Information Manual

A WORD ABOUT TRADEMARKS . . .

The following AT&T trademarks are used in this manual:

WE®
UNIX®

The following trademarks, owned by entities other than AT&T, are also used in this manual:

MS-DOS®
VMS®

FOREWORD

This manual contains detailed information regarding the design and applications of the *WE* DSP16/DSP16A Digital Signal Processor that is essential to engineers designing systems using this device. The *WE* DSP16 and DSP16A Support Software Library, *WE* DSP16 Development System, and the *WE* DSP16A Development System are available to aid in developing software for the devices and integrating them into system environments.

Additional information on the digital signal processor product line is available in the form of manuals, data sheets, and application notes.

WE DSP16 AND DSP16A DIGITAL SIGNAL PROCESSORS INFORMATION MANUAL

CONTENTS

APPENDIX C. DSP16/DSP16A INSTRUCTION SET SUMMARY

APPENDIX D. PROGRAMMABLE REGISTERS

GLOSSARY

LIST OF FIGURES

LIST OF TABLES

Chapter 1
Introduction

CHAPTER 1. INTRODUCTION

CONTENTS

1. INTRODUCTION

The *WE* DSP16 and *WE* DSP16A Digital Signal Processors are 16-bit, high-performance, CMOS integrated circuits. These devices can be programmed to perform a wide variety of signal-processing functions. They are the DSPs of choice for applications requiring low power, high performance, and low cost.

Both devices share the same architecture, instruction set, and I/O interfaces. The DSP16A offers greater speed and more internal memory for applications that require those characteristics.

This manual is a reference guide for the DSP16 and DSP16A devices. It describes the architecture, instruction set, and interfacing requirements of both devices.

1.1 FEATURES

The *WE* DSP16 Digital Signal Processor features:

- 55 or 75 ns instruction cycle
- 16 × 16-bit multiply/add in one instruction cycle
- Two 36-bit accumulators
- 2048 words of ROM, 512 words of RAM
- Complete set of ALU (arithmetic logic unit) operations
- Immediate, indirect, and compound addressing modes
- Instruction cache for high-speed, ROM-efficient vector operations
- Serial and parallel I/O ports with multiprocessor capability
- Low-power CMOS technology, 84-pin plastic chip-carrier package

In addition to the DSP16's features, the DSP16A offers:

- 25 or 33 ns instruction cycle
- 4096 words of ROM and 2048 words of RAM on-chip

1.2 DEVICE DESCRIPTION

1.2.1 Architecture

The arithmetic unit contains a 16 × 16-bit parallel multiplier that generates a full 32-bit product in one instruction cycle. The product can be accumulated with one of two 36-bit accumulators. The data in these accumulators can be directly loaded from or stored to memory in 16-bit words with automatic saturation on overflow. The ALU supports a full set of arithmetic and logic operations on either 16- or 32-bit data. A standard set of ALU conditions can be tested for conditional branches and subroutine calls. This procedure allows the processor to function as a powerful 16- or 32-bit microprocessor for logical and control applications.

Two addressing units support high-speed, register-indirect memory addressing with postmodification of the register. Four address registers can be used for either read or write addresses to the RAM without restrictions. One address register is dedicated to the ROM for table look-up. Direct and immediate addressing is supported at a cost of only one additional instruction cycle and one ROM location.

The DSP16 on-chip memory includes 2048 words of ROM and 512 words of RAM. For prototyping or for applications that require frequent program modification, the ROM can be replaced with up to 64 Kwords of external memory. The DSP16A offers twice as much ROM (4096 words) and 4 times as much RAM (2048 words) as the DSP16. The DSP16A ROM may be replaced with up to 64 Kwords of external memory or augmented with an additional 60 Kwords of external memory. An on-chip cache memory can be selectively used to store such repetitive operations as a filter section. The code in the cache can be repeated up to 127 times with no looping overhead. In addition, operations in the cache that require a ROM access (for example, reading fixed coefficients) execute at twice the normal rate. The cache greatly reduces the need for writing in-line repetitive code and, therefore, conserves ROM storage.

The device has both serial and parallel I/O ports. The serial I/O unit is double-buffered and easily interfaces with other DSP16 devices, commercially available codecs, and time-division multiplexed (TDM) channels with few (if any) additional components. The parallel I/O unit is capable of interfacing to a 16-bit bus containing other DSP16 devices, microprocessors, microprocessor peripherals, or other I/O devices.

The processor is implemented in low-power CMOS technology and is packaged in an 84-pin plastic leaded chip carrier (PLCC) or a 133-pin ceramic pin-grid-array (PGA) package.

1.2.2 Instruction Set

The DSP16/DSP16A instructions fall into five possible categories: multiply/ALU, special function, control, data move, and cache. All instructions are 16 bits wide and have a C-like assembler syntax. Although some pipelining of DSP16/DSP16A instructions is necessary to achieve the real-time performance required in many signal processing applications, the degree of pipelining has been reduced from previous generation DSPs to simplify programming. Latency effects have been eliminated.

1.3 APPLICATION DEVELOPMENT

Application development is aided by the use of the *WE* DSP16 and DSP16A Support Software Library, the *WE* DSP16 and DSP16A Application Library, and the *WE* DSP16/DSP16A Digital Signal Processor Development Systems.

1.3.1 Support Software Library

Support software tools to help create, test, and debug DSP16/DSP16A application programs are available from the *WE* DSP16 and DSP16A Support Software Library. The support software library consists of an integrated assembler and simulator that run on the *UNIX*®, *MS-DOS**, or VMS† Operating Systems.

The DSP16/DSP16A software simulator provides access to all registers and to memory and allows program breakpointing. The simulator also provides the user interface to the *WE* DSP16/DSP16A Development Systems. The hardware development systems have many of the breakpointing capabilities of the software simulator and can also be used in a simulator/accelerator mode to speed software simulations.

1.3.2 Development System

Application system hardware development and software testing are supported by the *WE* DSP16/DSP16A Development Systems. Each development system provides in-circuit emulation to facilitate real-time debugging of user hardware, as well as a simulator/accelerator to speed software simulations. Up to 16 development systems can be cascaded when developing applications that involve multiple DSPs.

While connected to the development system, the user may edit, assemble, and load programs, as well as utilize the software simulator. An assembled program can be transferred from the host into the development system's program memory through an internal bus of a parallel interface to the AT&T PC 6300 (or compatible) Personal Computer.

The software simulator and the development system can be used in three different modes:

- **Simulation Mode**. In this mode, the development system is not being used; program execution is being simulated in the host computer. This mode is used for program development and testing.
- **Hardware Mode**. The program has been downloaded into the development system and is being executed by the actual DSP16/DSP16A device. Hardware mode is used for real-time program testing.
- **Simulator/Accelerator Mode**. The program is executed in the development system (as in the hardware mode), but data is supplied to and from the host. The simulator/accelerator mode is used to speed algorithm development by executing the program in the hardware of the development system rather than in the host computer.

* Registered trademark of the Microsoft Corporation

† Registered trademark of the Digital Equipment Corporation

1.4 DOCUMENTATION

This document is a reference guide for the DSP16 and DSP16A devices. It describes the architecture, instruction set, and interfacing requirements of the both devices. Information on DSP16/DSP16A programming techniques and several complete sample application programs are provided. The remaining chapters of this manual are outlined below:

- **Chapter 2. DSP16/DSP16A Architecture** – a detailed description of the DSP16/DSP16A device, including separate sections describing the major elements of the architecture and how they function.
- **Chapter 3. Instruction Set** – lists the complete instruction set of both devices and provides a description of each instruction, including restrictions and normal uses. Addressing modes are also discussed in detail.
- **Chapter 4. Device Programming** – discusses topics related to programming beyond the syntax of the instruction set. Possible problems are discussed, along with solutions and advice.
- **Chapter 5. Serial I/O** – a detailed analysis of the operation of the serial I/O port. Information specific to the serial I/O unit is provided that is essential to designs that utilize this port.
- **Chapter 6. Parallel I/O** – a detailed analysis of the operation of the parallel I/O port. Information specific to the parallel I/O unit and detailed information regarding the interrupt mechanism are provided that are essential to designs utilizing the parallel port.
- **Chapter 7. Interface Guide** – provides information regarding the physical design of the devices, including pin assignments, electrical characteristics, external memory interfacing, and reset and interrupt control.
- **Appendix A. Instruction Set Encoding** – lists the hardware-level encoding of the instruction set.
- **Appendix B. Programming Examples** – presents four complete sample application programs that demonstrate proper programming techniques.

1.4.1 Other Applicable Documentation

When designing application hardware and software, it is important to have accurate information. A variety of documents exist to provide specific information on various members of the DSP16 product family. Contact your AT&T Account Manager for the latest issue of any of the following documents.

- The *AT&T Digital Signal Processor Family Description* introduces the AT&T DSP family of products and provides a brief overview of the capabilities of its members.
- *WE® DSP16 and DSP16A Digital Signal Processor Information Manual* (this manual) is a reference guide for the DSP16/DSP16A/DSP16-Military devices. It describes the architecture, instruction set, and interfacing requirements. Information on programming techniques and several complete sample application programs are provided.

- *WE® DSP16 Digital Signal Processor* Data Sheet provides up-to-date timing requirements and specifications, electrical characteristics, and a summary of the instruction set and device architecture.
- *WE® DSP16A Digital Signal Processor* Data Sheet provides up-to-date timing requirements and specifications, electrical characteristics, and a summary of the instruction set and device architecture.
- *WE® DSP16 and DSP16A Support Software Library User Manual* provides the information necessary to install and use the DSP16/DSP16A support software. This manual is also required when working with the DSP16 Development System or the DSP16A Development System, as the support software provides an interface between the host computer and the development system.
- *WE® DSP16 and DSP16A Development System User Manual* provides the information necessary to set up and use the DSP16 and DSP16A Development Systems.

1.5 ASSISTANCE

Assistance is available during conception, development, and throughout the life of the product. These services include:

- Technical documentation and product samples
- Information on determining and selecting the appropriate hardware and software
- The AT&T DSP Bulletin Board provides the latest and most up-to-date information about AT&T DSP products and application assistance:

  1200/2400 baud
  7 data bits, even parity
  1 stop bit
  201-834-6068

For technical assistance or further information including ordering information and part numbers, please contact the nearest office through the following phone numbers:

Domestic (USA & Canada)

Northeast Region 508-626-2161	Mid-Atlantic Region 215-768-2626	Southeast Region 404-446-4700
North Central Region 612-885-4300	South Central Region 214-869-2040	Southwest Region 602-244-1100
Rocky Mountain Region 303-850-2935	Pacific Northwest 408-522-5555 or 503-244-3883	Southern California 818-902-0139 or 714-220-6223

International

Europe (except Spain and Portugal) +49 89 950 86 0 Telfax: +49 89 950 86 111	Spain and Portugal +34 1 404 6012 Fax: +34 1 404 6252
Japan 813-593-3301 Telex: J32562 ATTIJ Fax: 813-593-3307	Pacific Rim 65-225-5233 Telex: RS 42898 Fax: 65-225-8725

Internal (AT&T Customers)

AT&T internal customers should contact their local AT&T Account Management Office. If the Account Management telephone number is not known, call **1-800-372-2447** and ask for the telephone number of your account representative.

Chapter 2
DSP16/DSP16A Architecture

CHAPTER 2. DSP16/DSP16A ARCHITECTURE

CONTENTS

2. DSP16/DSP16A ARCHITECTURE

The major elements of the DSP16/DSP16A architecture are the memories, ROM and RAM; the cache; the address arithmetic units; the data arithmetic unit; the I/O, serial and parallel; and the control unit that connects these elements in a pipeline. Figure 2-1 shows a block diagram.

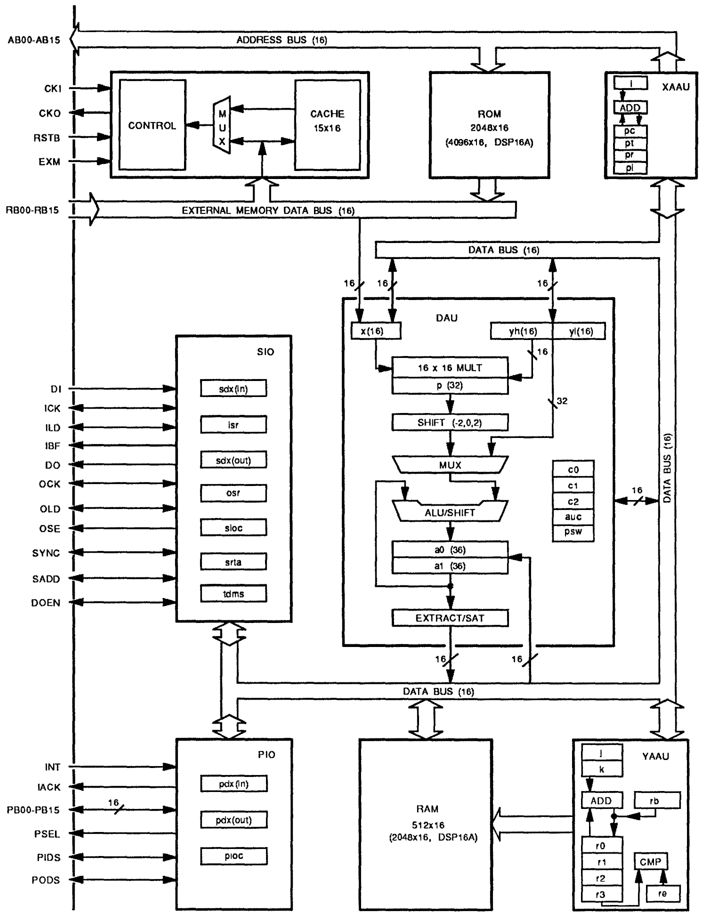


Figure 2-1. DSP16/DSP16A Digital Signal Processor Block Diagram

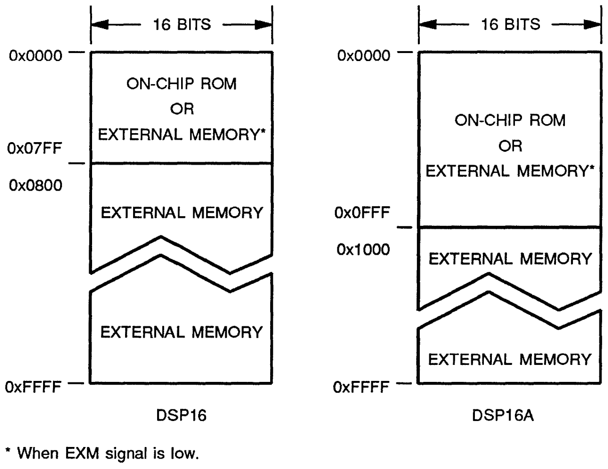


Figure 2-2. DSP16/DSP16A Memory Maps

2.1 MEMORY

The DSP16/DSP16A device provides both on-chip program memory and on-chip data memory. The program instructions and fixed operands are stored in the ROM. Instructions and immediate values are 16 bits wide and are fetched in one memory access each. Such variable operands as adaptive filter coefficients, state variables, intermediate results, and I/O data are stored in RAM. The on-chip ROM can be replaced with up to 64 Kwords of off-chip program memory on the DSP16. The DSP16A ROM may either be replaced with up to 64 Kwords or augmented with an additional 60 Kwords of off-chip program memory.

2.1.1 ROM

The DSP16 device provides a 2K × 16-bit, on-chip ROM for program memory and immediate data. The on-chip ROM for the DSP16 device can be replaced by off-chip memory. The 16-bit address and ROM data buses are available off-chip for external program memory expansion. The DSP16 device provides on-chip address bus and data bus latches and memory control signals to allow for a zero chip interface to memory devices. The DSP16 device can execute programs residing in either the 2 Kword on-chip ROM or in off-chip program memory (64 Kwords maximum). A memory map for DSP16 program memory is provided in Figure 2-2.

The DSP16A features a larger 4K × 16-bit, on-chip ROM. Like the DSP16, the ROM of the DSP16A can be replaced with up to 64 Kwords of external memory. The DSP16A also allows the use of both internal ROM and up to 60 Kwords of external memory.

2.1.2 RAM

The DSP16 device provides a 512 × 16-bit, on-chip, static RAM to store intermediate results of calculations and I/O data. The DSP16A provides a 2048 × 16 on-chip, static RAM. RAM expansion is supported by the parallel I/O unit (see Chapter 6).

2.2 CACHE

The on-board cache memory selectively stores repetitive operations to increase the throughput and the coding efficiency of the DSP16/DSP16A device. The cache can store up to 15 instructions at a time. The DSP16/DSP16A device can be programmed to execute the instructions in the cache up to 127 times without having to use branching instructions. Instructions previously stored in the cache can be re-executed without reloading the cache.

Cache instructions eliminate the overhead when repeating a block of instructions. Therefore, the cache reduces the need to implement a routine that uses in-line coding in order to maximize the throughput. A routine utilizing the cache uses less ROM locations than an in-line coding of the same routine.

For two-operand multiply/arithmetic logic unit (ALU) instructions that do not require a write memory, decreasing the execution time from two instruction cycles to one instruction cycle results in an increase in throughput.

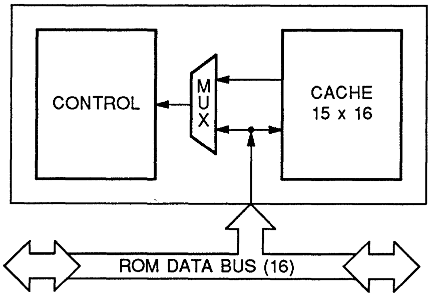


Figure 2-3. Control and Cache

2.3 CONTROL

The control block provides overall DSP16/DSP16A system coordination. The instructions are decoded by hardware in the control block. The execution of the phases of an instruction is controlled by hardware throughout the DSP16/DSP16A device. The hardware sequences instructions through the pipeline and controls the I/O, the processing, the memory accesses, and the timing necessary to perform each operation.

2.4 ADDRESS ARITHMETIC UNITS

Separate address arithmetic units for the on-chip ROM and RAM are provided. Each address arithmetic unit consists of static registers and an adder. Addresses are held by four of the registers in each address arithmetic unit. The remaining registers hold values that can be used to modify the address pointers. The adder is used to increment or decrement the addresses stored in the registers.

2.4.1 ROM Address Arithmetic Unit

The ROM address arithmetic unit (XAAU) consists of a 12-bit adder; a 12-bit static offset register, i; and four 16-bit static pointer registers: the program counter, pc; the program return, pr; the program interrupt, pi; and the table pointer, pt. These registers are used to address the ROM. The i register can be used to postmodify the pt register. The pt, pr, and i registers are user-accessible and can be modified under program control.

The pc register can be loaded with the address of a subroutine or branch directly from the control section. The program return address from a subroutine invoked using the call control instruction is saved in the pr register. The program return address from an interrupt is saved in the pi register. The pc register is loaded with the address in pr when returning from a subroutine or in pi when returning from an interrupt. The pc register can also be loaded from the pt register.

The pt register is normally used to point to tables of fixed data in ROM. The contents of the pt register can be modified by 1 or the value stored in the i register.

The i register contains a 12-bit, 2's complement signed number with a range of –2048 to +2047. The adder in the XAAU is used to postmodify the contents of the pt register. The data in the i register is sign-extended to 16 bits when transferred on the data bus. The XAAU has a 12-bit adder. Therefore, the address space can be viewed as sixteen 4 Kword pages. The adder is used to modify the 12 least significant bits (LSBs). The four most significant bits (MSBs) of the address may be changed by **goto pt** and **call pt** instructions.

Due to the XAAU 12-bit adder, pt can only be postincremented to 4095 by x = *pt++. But, since pt is a 16-bit unsigned register, it can be loaded with values to 64K.

The pi register is a 16-bit "shadow" register. Each time the pc register is modified, its new value is also loaded into the pi register. While in an interrupt service routine (ISR), this "shadowing" is disabled, and pi holds the last value of pc before the interrupt was taken. The return from interrupt instruction (**ireturn**) is simply a "goto pi" instruction.

The pi register may be read or written while in an ISR, but writing affects the return address. When not in an ISR, writing to pi has no effect on the contents of pi (the write to pi resets the pseudorandom sequence generator).

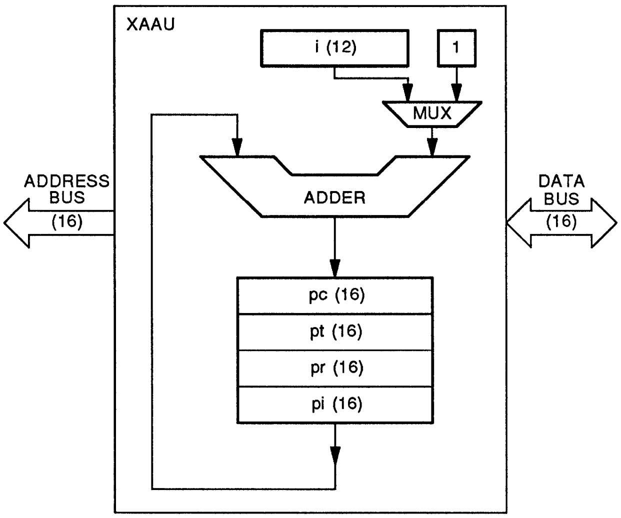
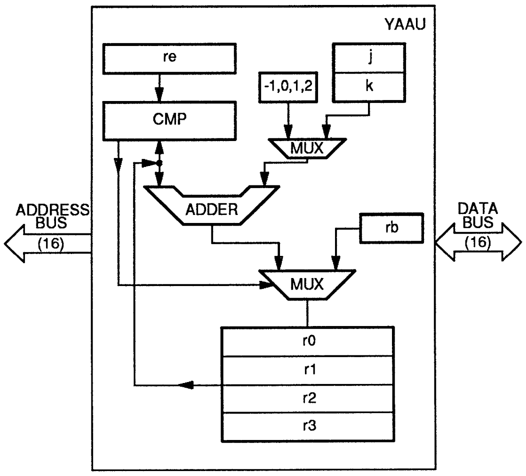


Figure 2-4. XAAU – ROM Address Arithmetic Unit



NOTE: All registers are 9 bits wide in the DSP16 and 16 bits wide in the DSP16A.

Figure 2-5. YAAU – RAM Address Arithmetic Unit

2.4.2 RAM Address Arithmetic Unit

The RAM address arithmetic unit (YAAU) consists of eight static registers and an adder. These registers are 9 bits wide in the DSP16 and 16 bits wide in the DSP16A. The RAM is addressed by the r0—r3 pointer registers. The j and k offset registers can be used to postmodify registers r0—r3. The rb and re registers are used when a register addressing the RAM is used in a cyclical (modulo) fashion. The eight YAAU registers are accessible to the user and can be loaded under program control.

The registers r0—r3 point to the RAM location that is the source of data to be loaded into the destination specified in the instruction or to the RAM location that is the destination of data from the source specified in the instruction. The r0—r3 pointers may be automatically postmodified by 0, +1, −1, +2, the contents of the j register, or the contents of the k register. The j and k registers contain 9-bit, 2's complement signed numbers with a range of −256 to +255 in the DSP16 and 16-bit, 2's complement signed numbers with a range of −32,768 to +32,767 in the DSP16A. The adder in the YAAU is used to postmodify the contents of the r0—r3 registers.

Data in RAM can be addressed by using a virtual shift addressing mode. This addressing mode forms the equivalent of a cyclic shift register within the RAM.

Virtual shift addressing realizes a shift register for the FIR filter tap values without moving the data stored in RAM. The memory space allocated in RAM for the table of data to be addressed by using virtual shift addressing is defined by the addresses loaded into rb and re. The rb contains the physical address of the first location of the shift register; re points to the last entry. When a pointer is used, its value is compared with the contents of re. If they are equal and the postincrement is 1, the RAM address arithmetic unit writes the value of rb into the RAM pointer after the memory access. The DSP16 device can support virtual shift addressing of shift registers of up to 512 words in length. The DSP16A device can support virtual shift registers of up to 2048 words in length. The virtual shift addressing mode is disabled when the value in re is 0. Register re is cleared (0) on reset.

2.5 DATA ARITHMETIC UNIT

The data arithmetic unit (DAU) is the main execution unit for signal processing algorithms. The DAU consists of a 16 × 16-bit multiplier, 36-bit ALU, and two 36-bit accumulators; a0 and a1. The DAU performs 2's complement, fixed-point arithmetic and is software configurable as a multiply/accumulate or ALU structure. The DAU multiplier and adder operate in parallel, each requiring one instruction cycle for their execution. Microprocessor-like instructions are executed by the ALU.

The multiplier executes a 16 × 16-bit multiply and stores the 32-bit product in the product register, p, in one instruction cycle. Data for the multiplier's inputs is stored in the 16-bit x register and the upper 16-bits (high half) of the 32-bit y register. The x register may be directly loaded from ROM, RAM, or the high half of an accumulator (bits 16—31 of the 36-bit word). The high half of the y register may be directly loaded from RAM or the high half of an accumulator.

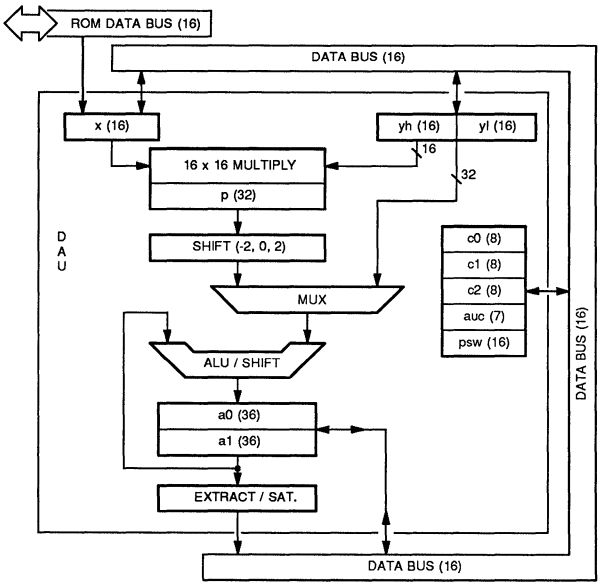


Figure 2-6. DAU – Data Arithmetic Unit

In addition to being used as an adder in the multiply/accumulate instructions, the 36-bit ALU provides the capability to implement functions and algorithms in the DSP16/DSP16A device that conventionally would have been executed in a microcomputer or a microprocessor. Operands to the ALU can be data in y, p, a0, or a1. The ALU sign-extends 32-bit operands from y or p to 36 bits and produces a 36-bit output (32 bits of data and 4 guard bits) in one instruction cycle. Either accumulator can receive the 36-bit result. The ALU supports diadic functions with register y and an accumulator, including addition, subtraction, and logical AND, OR, and XOR. Monadic functions of an accumulator include rounding, negation, incrementation, and left and right shifts of 1, 4, 8, or 16 bits.

The y register is 32 bits wide. To read or write the low half of the y register (bits 0—15), yl is used in an assembly-language instruction. When y is used in an assembly-language instruction, the DSP16/DSP16A device will read or write the high half (bits 16—31) of the y register. Automatic clearing of yl may be selected (according to the CLR field of the auc register) to simplify 16-bit operations. If clearing of the yl is enabled, the lower half of register y is cleared (0) with a write to the high half of the y register. Writes to yl do not change the data in the high half of y.

The accumulators are 36 bits wide. The contents of either the high half of the accumulator (bits 16—31) or the low half of the accumulator (bits 0—15) may be transferred to the 16-bit data bus. Automatic clearing of a0l or a1l may be selected (according to the CLR field of the auc register) to simplify 16-bit operations. If clearing of the low half of the accumulator is enabled, the lower half of the accumulator is cleared (0) with a write to the high half of the accumulator. Writes to the low half of the accumulator do not change the data in the high half of the accumulator. When the high half of an accumulator is loaded, the guard bits (35—32) are also loaded with the value of bit 31 and, thereby, sign-extended. Access to the guard bits for reading and writing is provided by the psw register.

The SAT field of the auc register allows the enabling of saturation on overflow when transferring the contents of accumulators onto the data bus. If saturation on overflow is enabled and either half of an accumulator is selected, the value transferred is saturated to 32 bits. If the selected accumulator has overflowed, then either the positive or negative (based on bit 35) saturation value is transferred.

$2^{31}-1$	is the positive saturation value
-2^{31}	is the negative saturation value.

A write of the contents of a 32-bit register to RAM requires two instructions: a write of the data in the high half of the register to RAM and a write of the data in the low half of the register to RAM. The order of the two writes to memory is left to the programmer. A read of the contents of RAM to a 32-bit register or accumulator also requires two instructions. If clearing of the low half of the destination's 32-bit register is enabled (according to the auc register, CLR field), the read of data in RAM to a 32-bit register must be done in the following order: load data to the high half of the register and load data to the low half of the register. This order is necessary because a load to the low half of a register does not change the data in the high half, while a load of the high half of a register clears the data from the low half. If clearing of the low half of the register is disabled, the two register loads may be performed in either order.

In addition to the registers already mentioned, the user has access to the arithmetic unit control register, auc; the processor status word register, psw; and the counters, c0—c2. The auc register configures some features of the data arithmetic unit. The psw register contains status information regarding the data arithmetic unit. Table 2-1 shows the contents of the psw register. The c0—c2 counters are 8 bits wide and may be used under program control to count events such as the number of times the program has executed a sequence of code.

Table 2-1. Processor Status Word (psw) Register

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Field	DAU Flags				X	X	a1[V]	a1[35—32]				a0[V]	a0[35—32]			

Bit(s)	Field	Value*	Result/Description
15—12	DAU Flags	Wxxx	LMI – logical minus when set.
		xWxx	LEQ – logical equal when set.
		xxWx	LLV – logical overflow when set.
		xxxW	LMV – mathematical overflow when set.
11, 10	X	—	Reserved.
9	a1[V]	W	Accumulator 1 (a1) overflow when set.
8—5	a1[35—32]	Wxxx	Accumulator 1 (a1) bit 35.
		xWxx	Accumulator 1 (a1) bit 34.
		xxWx	Accumulator 1 (a1) bit 33.
		xxxW	Accumulator 1 (a1) bit 32.
4	a0[V]	W	Accumulator 0 (a0) overflow when set.
3—0	a0[35—32]	Wxxx	Accumulator 0 (a0) bit 35.
		xWxx	Accumulator 0 (a0) bit 34.
		xxWx	Accumulator 0 (a0) bit 33.
		xxxW	Accumulator 0 (a0) bit 32.

* W indicates that the bit may be read or written.

2.5.1 Arithmetic and Precision

Fixed-point, 2's complement arithmetic is used throughout the DSP16/DSP16A devices. The arithmetic bit alignment for the DSP16/DSP16A devices is shown in Figures 2-7, 2-8, and 2-9. The 16-bit data in the x register and in the high half of the y register can be multiplied together and the 32-bit result is stored in the p register. The data in the y or p registers can be operated on by the ALU and the result stored in either of the 36-bit accumulators. The 32-bit data from the y or p register is sign-extended to 36 bits when operated on by the ALU.

For notational convenience, the 36-bit accumulators can be thought of as having an implied binary point to the right of bit 16. Bits 15—0 are then the fractional part, which is referred to as aNl, where aN = a0, or a1, and bits 35—16 are the integer part. The ALU operates on all 36 bits of the accumulators. The CLR field of the auc register controls automatic clearing of the low half of a0, a1, and y, making it easy to perform 16-bit integer operations in the ALU by automatically clearing the low half of the register when the high half is loaded.

The data transferred between an accumulator and memory or register must be scaled properly to reflect the bit alignment between p and a[0, 1] determined by the auc word. The user can select where the data in p is placed in the accumulator. Table 2-2 shows how two bits in the auc register, auc[1, 0], determine the bit alignment of the data in p with respect to the data in the accumulators. The connection of the data bus to the RAM, the accumulators, and the remaining registers in the DSP16/DSP16A device is fixed.

Table 2-2. Arithmetic Unit Control (auc) Register

Bit	6	5	4	3	2	1	0
Field	CLR			SAT		ALIGN	

Bit(s)	Field	Value	Result/Description
6—4	CLR	1xx	Clearing yl is disabled (enabled when 0).
		x1x	Clearing a1l is disabled (enabled when 0).
3, 2	SAT	1x	a1 saturation on overflow is disabled (enabled when 0).
		x1	a0 saturation on overflow is disabled (enabled when 0).
1, 0	ALIGN	00	$p \leftarrow (x \times y)$.
		01	$p \leftarrow (x \times y) \div 4$.
		10	$p \leftarrow (x \times y) \times 4$.
		11	Reserved.
		xx1	Clearing a0l is disabled (enabled when 0).

Note: The auc register is not affected by reset.

If the auc[1, 0] bits are 00, the data in the p register is not shifted with respect to the bits in the accumulator before p[31—0] is transferred into bits 31—0 of an accumulator. The sign of p is extended by four bits, p[35—32], to provide overflow protection. The data transfer from p to an accumulator moves the overflow bits of the product into the guard bits 35—32 of the accumulator (see Figure 2-7 for the bit alignment in the DAU for auc[1,0]=00). This mode is most often used when both x and y operands are 16-bit integers.

If the auc[1, 0] bits are 10, the data in the p register is shifted two bits to the left with respect to the bits in the accumulator before p[31—0] are transferred into bits 33—2 of an accumulator. Bits 1 and 0 of the accumulator are not changed by the load of the accumulator with the data in p, since 00 is added to or copied into these accumulator bits as indicated in Figure 2-8. The sign of p is extended by two bits, p[33—32], to provide overflow protection. The data transfer from p to an accumulator moves the overflow bits of the product into guard bits 35—34 of the accumulator (see Figure 2-8 for the bit alignment in the DAU for auc[1,0]=10). This mode is often used in filtering applications where coefficients in the x register are in Q14 format (2 magnitude bits, 14 fractional bits), and state variables in the y register are 16-bit integers. If the p register is not shifted prior to accumulation, the accumulated result would have 4 guard bits, 18 magnitude bits, and 14 fractional bits. Since it is often desirable to have the implied binary point to the right of bit 16 (16 fractional bits), the setting auc[1,0]=2 automatically shifts the result 2 bit locations to the left generating an accumulated result with 4 guard bits, 16 magnitude bits, and 16 fractional bits.

If the auc[1,0] bits are 01, the data in the p register is shifted two bits to the right with respect to the bits in the accumulator before p[31—2] are transferred into bits 29—0 of an accumulator. Bits p[1,0] are not saved in the accumulator by the load of the accumulator with the data in p in this auc setting. The sign of p is extended by six bits, p[37—32], to provide overflow protection. The data transfer to an accumulator from p moves the overflow bits of the product into the guard bits 35—32 and bits 31—30 of the accumulator (see Figure 2-10 for the bit alignment in the DAU for auc[1,0]=01). This setting is most useful when avoiding overflow is a primary consideration, and the loss of the two LSBs of the product can be tolerated.

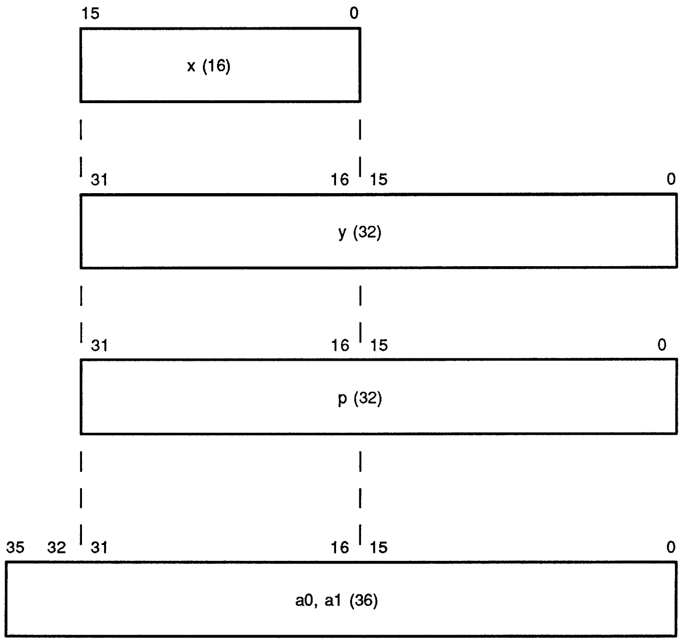


Figure 2-7. DSP16/DSP16A Arithmetic Bit Alignment When auc[1,0]=00

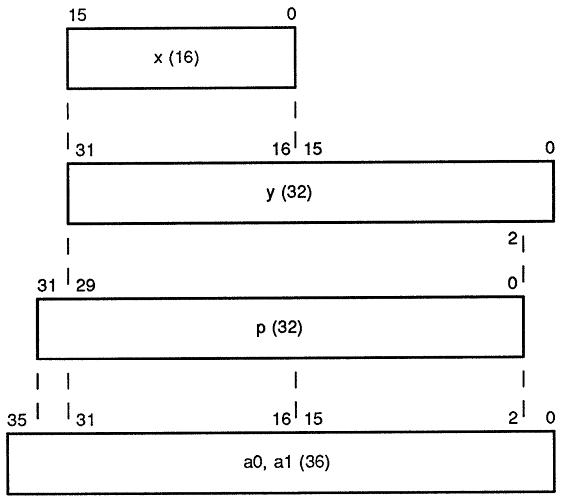


Figure 2-8. DSP16/DSP16A Arithmetic Bit Alignment When auc[1,0]=10

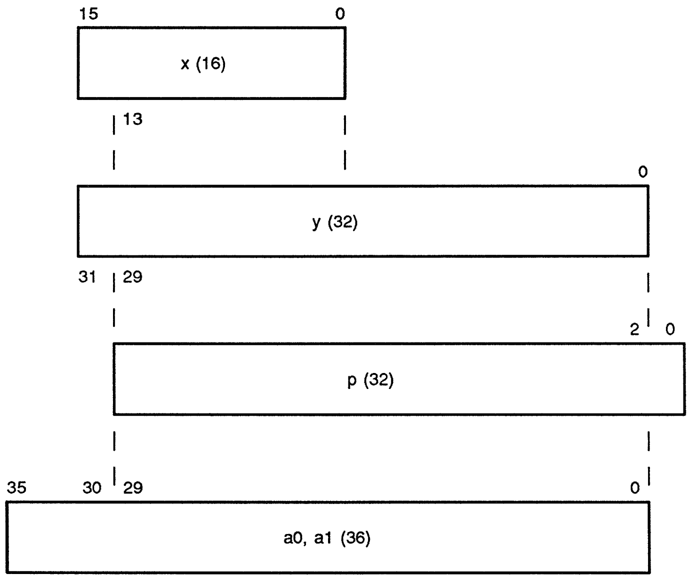


Figure 2-9. DSP16/DSP16A Arithmetic Bit Alignment When auc [1,0]=01

2.6 SERIAL I/O

The serial I/O port (SIO) allows the DSP16/DSP16A device to interface serially to other devices with few, if any, external chips (see Figure 2-10). The serial I/O port converts the serial input data stream to a parallel input data word and the parallel output data word to a serial output data stream. Serial I/O transfers are double-buffered to enable the DSP16/DSP16A device to handle back-to-back serial transfers. The second serial transmission can begin before the data from the first serial transmission has been processed. The external DSP16/DSP16A serial I/O control signals allow a zero chip interface to commercially available codecs and to the DSP16/DSP16A, DSP32, and DSP32C devices for multiple DSP applications.

The serial I/O control register, sioc, allows the specification under program control of: the length of the serial input and output data words; the mode, active or passive, of the serial bit clocks; the mode, active or passive, of the serial load signals; bit ordering of the I/O; the active serial I/O bit clock rate; and active load generated from either ICK or OCK. The length of serial input and output data words can be 8 or 16 bits. The serial I/O bit clocks and the serial I/O load signals are transmitted by the DSP16/DSP16A device to the rest of the external system in the active mode or are generated by the external system and transmitted to DSP16/DSP16A device in the passive mode. The mode of the input bit clock and load signal can be selected independent of the mode for the output bit clock and load signal. Active serial I/O bit clock rates of CKI/4, CKI/12, CKI/16, and CKI/20 can be selected. The bit order of the serial input data can be specified to be bit-reversed when the input is moved from the serial input buffer to the destination register or RAM location; the bit order of the data can be specified to be bit-reversed when data is transferred to the serial output buffer from the source register or RAM location. Bit reversal of the data is necessary for μ-law and A-law conversion and is performed in hardware rather than in software. See Chapter 5.

The tdms register specifies the time slot of the DSP in a time-division multiplexed signal, whether the DSP is operating in a single or multiple DSP16/DSP16A environment and the active frame synchronization signal clock rate, f/128 or f/256.

The synchronization clock can be active or passive. In the multiple DSP environment, a DSP16/DSP16A device can directly address a maximum of seven other DSP16/DSP16A devices via the serial I/O port. The 16-bit srta register is loaded with two 8-bit addresses. One address specifies the receiving DSP identity; the second address identifies the destination DSP, where the serial data is accepted. See Chapter 5 for more detailed information.

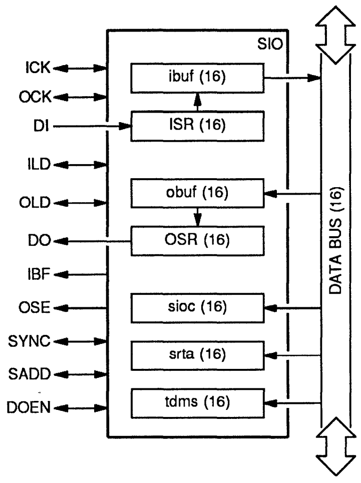


Figure 2-10. SIO – Serial Input/Output Unit

2.7 PARALLEL I/O

The DSP16/DSP16A parallel I/O port (PIO) provides a 16-bit, bidirectional data link to microprocessors and other I/O devices (see Figure 2-11). The parallel I/O port supports bidirectional communication with other devices over a wide range of data transfer rates.

The parallel I/O control word, pioc, allows the specification, under program control, of the configuration of the PIO data pins; the mode, active or passive, of the parallel I/O data strobe signals; and the width of the parallel I/O data strobe signals in the active mode. The 16-bit PIO data bus can be configured to transmit and receive 8 bits simultaneously or, under program control, either to input 16 bits or output 16 bits. The parallel I/O data strobe signals are transmitted by the DSP16/DSP16A device to the external system in the active mode or are generated by the external system and transmitted to the DSP16/DSP16A device in the passive mode. The mode of the input data strobe signal can be selected independent of the mode for the output data strobe signal. In the active mode, the pulse width of the strobe signals can be selected. The contents of the pioc register can be read to determine the status of the DSP16/DSP16A serial and parallel I/O ports. The pioc register contains a bit field with the status of the serial and parallel I/O flags. Under program control, this bit field can be read to determine the state of the I/O flags; the program cannot write this bit field in the pioc register.

The DSP16/DSP16A device can directly address two devices via the PIO ports, pdx0 and pdx1. The signal level on the PIO address line, psel, is low when data is written to pdx0 and high when data is written to pdx1. A DSP16 instruction writes to either pdx0 or pdx1 to output data via the PIO port. A DSP16 instruction reads pdx0 or pdx1 to access the data input to the PIO port by external devices. The PIO data strobe signals are asserted when data is written to the PIO port by the DSP16/DSP16A device or by external devices. The PIO port can be used to interface the DSP16/DSP16A device with a large external RAM memory.

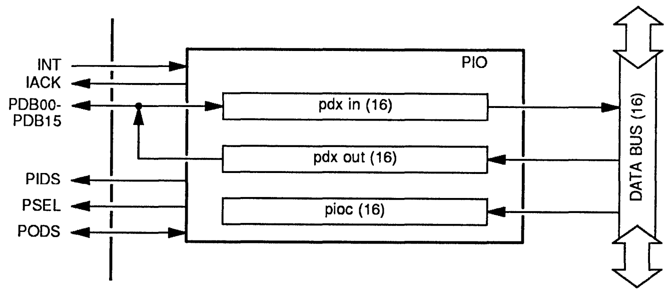


Figure 2-11. PIO – Parallel Input/Output Unit

2.8 INTERRUPTS

The DSP16/DSP16A device has one external interrupt request signal and four internal interrupt request signals. Hence, it can respond to one external and four internal conditions. These conditions are:

- **INT – Interrupt by an External Device.** An external device has requested service by asserting the INT pin.
- **IBF – Input Buffer Full.** Indicates that an external device has written data into the device's serial input buffer.
- **OBE – Output Buffer Empty.** Indicates that an external device has read data from the SIO serial output buffer.
- **PIDS – Parallel Input Data Strobe.** Indicates that an external device has written data into the PIO parallel input register.
- **PODS – Parallel Output Data Strobe.** Indicates that an external device has read the data from the PIO parallel output buffer.

Each interrupt is maskable; that is, by appropriately clearing bits in the pioc register, the associated interrupt is ignored. The interrupt mask bits, pioc[5—9], are cleared (0) on reset, thereby disabling all interrupts.

An interrupt causes a **goto 1** instruction to be jammed into the instruction register. When in an interrupt service routine, the shadowing of the pc register is disabled (see Section 2.4.1). The DSP16/DSP16A device also provides a software interrupt facility. The instruction **icall** causes the same sequence of events to occur as any other interrupt source, except the branch address of the interrupt service routine is 2 rather than 1. Note that the branch address for interrupts is always at location 1 or 2 (the branch is not limited to the current 4 Kword page).

Table 2-3. Parallel I/O Control (pioc) Register

Bit	15	14 13	12	11	10	9—5	4—0
Field	IBF	STROBE	PODS	PIDS	S/C	INTERRUPTS	STATUS

Bit(s)	Field	Value*	Result/Description
15	IBF	R	IBF interrupt status bit (same as bit 4).
14, 13	STROBE		Strobe width of **PODS PIDS**
		00	T† T
		01	2T 2T
		10	3T 3T
		11	4T 4T
12	PODS	0	PODS is an input (passive mode).
		1	PODS is an output (active mode).
11	PIDS	0	PIDS is an input (passive mode).
		1	PIDS is an output (active mode).
10	S/C	0	Not S/C mode.
		1	S/C mode.
9—5	INTERRUPTS	1xxxx	IBF interrupt is enabled (disabled when 0).
		x1xxx	OBE interrupt is enabled (disabled when 0).
		xx1xx	PIDS interrupt is enabled (disabled when 0).
		xxx1x	PODS interrupt is enabled (disabled when 0).
		xxxx1	INT interrupt is enabled (disabled when 0).
4—0	STATUS	Rxxxx	IBF status bit.
		xRxxx	OBE status bit.
		xxRxx	PIDS status bit.
		xxxRx	PODS status bit.
		xxxxR	INT status bit.

* R indicates a read-only bit.
† T = 2 × tCKIHCKIH

2.8.1 Hardware Description

The external interrupt mechanism has two relevant signals: INT and IACK.

- **INT – Interrupt Request.** When asserted, this input indicates to the DSP16/DSP16A device that an external device is requesting service. To guarantee that this request is serviced, the INT signal must remain asserted for twice the period of the DSP16/DSP16A device's CKO signal (i.e., 4 × tCKIHCKIH).

- **IACK – Interrupt Acknowledge.** When asserted, this output indicates that one of the internal or the external interrupt requests has been recognized by the DSP16/DSP16A device. Once the IACK signal has been asserted, it is negated upon completion of the interrupt service routine (i.e., upon execution of an ireturn instruction). IACK, when active, masks the interrupt request signal (INT).

2.8.2 Software Description

After recognizing an interrupt, the DSP16/DSP16A device completes execution of the current instruction and then branches to address 1. Branch and conditional branch instructions and instructions executing within the cache are not interruptible. If an interrupt request occurs during an uninterruptible instruction, the DSP16/DSP16A device does not recognize the request until an interruptible instruction is encountered. See Chapter 3 for more information on the DSP16/DSP16A instruction set.

The interrupt service routine entered should clear the respective status bit in the pioc register. If this is not done, the DSP16/DSP16A device repeatedly recognizes the interrupt until the appropriate status bit in the pioc is cleared. (This can be useful in certain applications.)

Individual status bits are cleared in the following manner:

- **IBF – Input buffer full** (pioc bits 4 and 15) is cleared by reading sdx, the serial input buffer.
- **OBE – Output buffer empty** (pioc bit 3) is cleared by writing to sdx, the serial output buffer.
- **PIDS – Parallel input data strobe** (pioc bit 2) is cleared by reading the parallel input buffer, pdx0 or pdx1.
- **PODS – Parallel output data strobe** (pioc bit 1) is cleared by writing to the parallel output buffer, pdx0 or pxd1.
- **INT – Interrupt by an external device** (pioc bit 0) is cleared when IACK makes a high-to-low transition (initiated by an **ireturn** instruction), indicating the end of the interrupt service routine.

The interrupt status bits are also cleared on reset.

See Section 6.3 for more information on interrupts utilizing the PIO and Section 4.2.5 for techniques on handling multiple interrupts.

Chapter 3

DSP16/DSP16A Instruction Set

CHAPTER 3. DSP16/DSP16A INSTRUCTION SET

CONTENTS

3. DSP16/DSP16A INSTRUCTION SET

All DSP16/DSP16A instructions are 16 bits wide and have a C-like syntax. Pipelining of the instructions is necessary to achieve the real-time performance required by many signal-processing applications. To facilitate programming, the degree of pipelining in the DSP16/DSP16A device has been reduced and the latency effects present in previous generation DSPs have been eliminated. The instructions fall into one of five possible categories:

- **Multiply/ALU** instructions are the primary instructions used to implement signal-processing programs. These instructions perform multiply/accumulate, logical, and other ALU functions and also transfer data between memory and registers in the data arithmetic unit.
- **Special Function** instructions are used to perform such operations as rounding, negation, and logical left shifts and arithmetic right shifts of accumulators. Special function instructions may be conditionally executed on the basis of the state of internal flags.
- **Control** instructions are used to control program flow. The **call, goto**, and **return** instructions are provided and may be conditionally executed on the basis of the state of internal flags.
- **Data Move** instructions are used to transfer data between registers, memory, and accumulators. Immediate loads of certain registers are also possible.
- **Cache** instructions allow the implementation of low overhead loops by loading a set of multiply/ALU and special function instructions into a cache memory and repetitively executing them (up to 127 times).

The following sections describe in detail the notation used in the instruction set, the addressing modes supported, the internal flags used by conditional instructions, and the five groups of instructions.

Note: Only multiply/ALU and special function instructions set DAU flags.

3.1 NOTATION

The following operators are used to describe the instruction set:

Operator	**Meaning**
*	$16 \times 16 \rightarrow$ 32-bit multiplication (Denotes register-indirect addressing when used as a prefix to an address register)
+	36-bit addition
−	36-bit subtraction
++	Register postincrement
−−	Register postdecrement
>>	Arithmetic right shift
<<	Logical left shift

&	32-bit bitwise AND
|	32-bit bitwise OR
^	32-bit bitwise EXCLUSIVE OR
:	Compound addressing

For all instructions listed in this chapter, the following are true:

- Brackets, [ ], are not part of the instruction syntax, but indicate that the enclosed item is optional.
- Parentheses, ( ), and braces, { }, are part of the instruction syntax and must appear where shown in the instruction.

The valid instruction groups for the DSP16/DSP16A device are represented in Tables 3-3 to 3-12. The items in Tables 3-3 to 3-12 and 3-14 to 3-23 that are written in lower-case letters are proper statements and must appear where shown in the instruction. The items with capital letters are not proper statements and are replaced with immediate data, a register name, or a condition.

3.2 ADDRESSING MODES

The DSP16/DSP16A Digital Signal Processor allows immediate, indirect, and compound addressing modes. Instructions using indirect and compound addressing are typically used to encode real-time, signal-processing algorithms and, hence, require less program memory and execute faster than immediate addressing.

3.2.1 Immediate Addressing

In immediate addressing, the operand is supplied in the instruction. This situation is useful when initializing registers and is provided at the expense of one additional ROM location and one instruction cycle of execution time. A short immediate addressing mode is supplied to set the YAAU registers, r0—r3, j, k, rb, and re which are 9 bits wide on the DSP16. The DSP16A YAAU registers are 16 bits wide, so short immediate addressing may only be used when loading values that are 9 bits long or less. Short immediate instructions execute in one cycle, use one ROM location, and are cacheable.

3.2.2 Indirect Addressing

Indirect addressing allows a register to be used as a pointer to another location. The terms X and Y specify the source of data from memory to registers or the destination of data from registers to memory:

X = *pt++ or *pt++i

Y = one of: *rM, *rM++, *rM– –, *rM++j

Note: M = one of: 0, 1, 2, 3

The term X represents the ROM data to be copied into the x register. The term Y represents the RAM data to be copied into the specified register or the data written to RAM from a register. The mnemonics for X and Y indicate register indirect addressing with a postmodification of the address pointer. The asterisk preceding the RAM or ROM address register stands for "the data pointed to by the address in the register." The mnemonics have the following meaning:

- ***r*M**. This example means "the data pointed to by the address in the register." The contents of the register are not altered by the operation.
- ***r*M*++, pt++**. The "++" following the address register indicates a postincrement of the address register. This example means "the data pointed to by the address in the register; add 1 to the contents of the register after the operation is complete."
- ***r*M*==**. The "==" following the address register indicates a postdecrement of the address register. This example means "the data pointed to by the address of the register: subtract 1 from the contents of the register after the operation is complete."
- ***r*M*++j**. The "++j" following the address register indicates a postincrement of the address register. This example means "the data pointed to by the address in the register; add the value of register j to the contents of the address register after the operation is complete." Negative values of j yield a postdecrement.
- ***pt++i**. The "++i" following the address register indicates a postincrement of the address register. This example means "the data pointed to by the address in the register; add the value of register i to the contents of the address register after the operation is complete." Negative values of i yield a postdecrement.

Modulo (virtual shift) addressing uses indirect addressing to form the equivalent of a cyclic shift register within the RAM. Addresses loaded into registers rb and re define the first and last physical addresses of the modulo, respectively. When a register is used as a memory pointer, its value is compared with re. If its value is equal to the contents of re and the postincrement is +1, then the value in rb is copied into the register after the memory access is complete. See Section 4.2.3.

3.2.3 Compound Addressing (Swapping).

Compound addressing is a memory read/write operation using only one pointer register. The term Z specifies a source and a destination for a compound RAM read followed by a write sequence. The mnemonics for Z are a shorthand notation for the compound addressing functions explained below and shown in Table 3-1. The term *temp* used in the descriptions is a hypothetical register used for illustration only.

Table 3-1. Compound Addressing Instructions

Instruction	Operations		
Z: R	Step 1	Step 2	Step 3
*rMzp:R	temp=R;	R=*rM;	*rM++=temp;
*rMpz:R	temp=R;	R=*rM++;	*rM=temp;
*rMm2:R	temp=R;	R=*rM—;	*rM++2=temp;
*rMjk:R	temp=R;	R=*rM++j;	*rM++k=temp;

Notes:
M can be 0, 1, 2, 3.
R can be one of x, y, yl, r0, r1, r2, r3, pt, pr, i, j, k, c0, c1, c2, rb, re, psw, auc, sioc, srta, sdx, tdms, pioc, a0, a0l, a1, a1l.
R and rM must not be the same register (i.e., r1pz:r1).

As with other instructions that use the y, a0, and a1 registers, the following rules apply when using the compound addressing mode:

- If clearing of the low half of the register is enabled (according to the CLR field of the auc register), the low half of the register is cleared when the high half is loaded.
- If saturation on overflow is enabled (according to the SAT field of the auc register), the value in the accumulator is limited. See Section 2.5.1.

Virtual shift addressing may be used with compound addressing. The contents of the address register are compared with the contents of register re during both the read and write cycles. If the contents of the address register are equal to the contents of re during the read cycle and the "*rMpz" mode is specified, r*M* is loaded with the contents of rb. If the contents of the address register are equal to the contents of re during the write cycle and the "*rMzp" mode is specified, r*M* is loaded with the contents of rb. See Section 4.2.3.

3.3 PROCESSOR FLAGS

Control and special function instructions may be conditionally executed on the basis of internal flags set by the previous ALU operation, the condition of one of the counters, or the value of a randomly set bit in the device. Multiply/ALU function statements and special function instructions affect the flags; loading an accumulator with a multiply/ALU transfer statement or a data move instruction does not affect the flags. The processor flags and their meanings are:

LMI **Logical Minus** – A logical minus is determined by the state of bit 35 of the last DAU operation result. If bit 35=1, the result is a negative number and LMI is true.

LEQ **Logical Equal** – A logical equal is determined by the sum of bits 35—0 of the last DAU operation result. If the sum of the bits equals zero, the result is zero and LEQ is true.

LLV **Logical Overflow (36-Bit Overflow)** – LLV is true if the sign of the result of an operation cannot be represented in a 36-bit accumulator.

LMV **Mathematical Overflow (32-Bit Overflow)** – LMV is true if the overflow bits (35—31) of the accumulator used in the last DAU operation are not identical. This indicates a number not representable in 32 bits.

Table 3-2 shows the mnemonics that are used in conditional instructions and their meanings. The state of the internal flags that causes the condition to be true is enclosed in parentheses after the description. For example, when testing the condition le, the result is true if either the logical minus (LMI) or logical equal (LEQ) flags are true.

Table 3-2. Conditional Mnemonics

Test	Meaning	Test	Meaning
pl	Result is nonnegative (not LMI).	mi	Result is negative (LMI).
eq	Result is equal to zero (LEQ).	ne	Result is not equal to zero (not LEQ).
gt	Result is greater than zero (not LMI and not LEQ).	le	Result is less than or equal to zero (LMI or LEQ).
lvs	Logical overflow set (LLV).	lvc	Logical overflow clear (not LLV).
mvs	Mathematical overflow set (LMV).	mvc	Mathematical overflow clear (not LMV).
c0ge*	Counter 0 greater than or equal to zero.	c0lt*	Counter 0 less than zero.
c1ge*	Counter 1 greater than or equal to zero.	c1lt*	Counter 1 less than zero.
heads†	Pseudorandom sequence bit set.	tails†	Pseudorandom sequence bit clear.
true	The condition is always satisfied in an if instruction.	false	The condition is never satisfied in an if instruction.

* Testing each of these conditions increments the respective counter being tested.

† The heads or tails condition is determined by a randomly set or cleared bit, respectively. The bit is randomly set with probability of 0.5. The random bit is generated by a 10-state pseudorandom sequence generator that is updated after either a heads or tails test. The pseudorandom sequence may be reset by writing any value to the pi register. Writing to the pi register does not affect the contents of the pi register except while in an interrupt service routine. A random rounding function can be implemented by using either of these two conditions.

3.4 MULTIPLY/ALU GROUP

The multiply/ALU instructions are the primary instructions used to implement signal-processing programs. Statements from this group can be combined to generate multiply/accumulate, logical, and other ALU functions and to transfer data between memory and registers in the data arithmetic unit. In the examples presented, the statements should be read from right to left, top to bottom. Statements within a multiply/ALU instruction are executed essentially in parallel. The multiply/ALU instructions usually consist of more than one part. Each part of an instruction is called a statement. The general rule is that valid instructions can be formed by choosing one statement from each statement column in Table 3-3. If either statement is not required, then a single statement from either column also constitutes a valid instruction. Conversely, valid instructions can be decomposed into separate statements, with each coming from a different column in the Table 3-3.

The multiply/ALU instructions consist of two parts: a function and a transfer (see Table 3-3). The statements in the function column can be separated into two types: those involving the multiplier and those involving only the ALU in the data arithmetic unit. The multiply/accumulate instructions typically used in signal-processing applications are assembled from statements from the function column that include the multiplication of the data in x and y[31—16]. In a multiply/accumulate instruction, the x and y registers are loaded with the operands, the product of the previous operands is generated, and the previous product is accumulated in a0 or a1.

The following example shows how a typical multiply/accumulate sequence is implemented.

Example:

```
Instruction #
    (1)                          y=Y    x=X
    (2)                 p=x*y
    (3)      aD=aS+p
```

In the example presented, the data in the X source is copied into the x register and the data in the Y source into bits 31—16 of the y register in line 1. In line 2, the product of the data in x and y[31—16] is generated and stored in p. In line 3 the data in the source accumulator, aS, and the data in p are added and the result loaded into the destination accumulator. Note that lines 2 and 3 could also have specified memory transfer operations for later instructions.

The ALU instructions perform one of the following:

- The logical operations of AND, OR, or XOR between an accumulator and the data in the y register.
- The addition or subtraction of the data in the y register from an accumulator.
- The load of an accumulator with the data in the y register.

The y register must be loaded prior to the ALU operation.

The following example shows how a typical logical operation is implemented.

```
(1)                 y=Y
(2)    aD=aS&y
```

In this example, the data in the Y source is copied into the y register in line 1. In line 2, the logical AND of the data in the source accumulator, aS, and the data in y as a result of line 1 are calculated and the result is loaded into the destination accumulator.

All multiply/ALU instructions require 1 word of memory. The number of instruction cycles required to execute an instruction in the multiply/ALU group is a function of the statement selected from the transfer column in Table 3-3. Instructions with statements in the transfer column involving a write to RAM are executed in two instruction cycles whether the instruction is in or out of the cache. Instructions with statements in the transfer column involving a read from the RAM and the ROM simultaneously are executed in two instruction cycles if not in the cache and one instruction cycle if in the cache. An instruction with no transfer statement executes in one instruction cycle either in or out of the cache. The remaining instructions are executed in one instruction cycle either in or out of the cache. Table 3-3 gives the number of instruction cycles for each case. The multiply/ALU instructions use one ROM location.

The no operation (**nop**) instruction is a special-case encoding of a multiply/ALU instruction and is executed in one instruction cycle. The assembly-language notation representation of a no operation instruction is either **nop** or a single semicolon (**;**).

Note that the function statements and transfer statements in Table 3-3 are chosen independently. Any function statement may be combined with any transfer statement to form a valid multiply/ALU instruction.

Table 3-3. Multiply/ALU Instructions

Function Statements		Transfer Statements		Transfer Cycles Out/In Cache
	p=x*y	y=Y	x=X	2/1
aD=p	p=x*y	y=aT	x=X	2/1
aD=aS+p	p=x*y	y[l]=Y		1/1
aD=aS−p	p=x*y	aT[l]=Y		1/1
aD=p		x=Y		1/1
aD=aS+p		Y		1/1
aD=aS−p		Y=y[l]		2/2
aD=y		Y=aT[l]		2/2
aD=aS+y		Z:y	x=X	2/2
aD=aS−y		Z: y[l]		2/2
aD=aS&y		Z: aT[l]		2/2
aD=aS|y				
aD=aS^y				
aS−y				
aS&y				

Table 3-4. Replacement Table for Multiply/ALU Instructions

Replace	Value	Meaning
aD, aS, aT	a0, a1	One of two DAU accumulators.
X	*pt++,*pt++i	ROM location pointed to by pt. pt is postmodified by +1 and i, respectively.
Y	*rM, *rM++, *rM—, *rM++j	RAM location pointed to by rM. (M= 0, 1, 2, 3). rM is postmodified by 0,+1,−1, and j, respectively.
Z	*rMzp, *rMpz, *rMm2, *rMjk	Read/write compound addressing. rM (M = 0, 1, 2, 3) is used twice. First, postmodified by 0, +1, −1, and j respectively and second, postmodified by +1, 0,+ 2, and k, respectively.

On the basis of the information given in Table 3-4, apply the following information to the function and transfer statements in Table 3-3:

- Loads of a0, a1, and y clear the lower half of the selected register when the appropriate CLR field bits in the auc register are zeroed.
- Loads of a0l, a1l, and yl do not change the data in the high half of the selected register.
- The y and p operands are sign-extended to match the width of the accumulators.

3.4.1 Function Statements

In the execution of these statements, the width of the number is extended to 36 bits, which is the size of the accumulators. This extension is accomplished by extending the sign bit in the p register to retain the correct 2's complement value. The multiplier performs a 2's complement multiply, using x and the high half of y (bits 31—16).

The statements must be written in the exact format shown. If the statements are written in any other way, for example, aD=p+aS instead of aD=aS+p, the assembler produces an error message.

- **p=x*y.** The contents of the x and the y (bits 31—16) registers are multiplied and the result is placed in the p register.
- **aD=p p=x*y**. The contents of the p register are copied into the destination accumulator, aD. The contents of the x and the y (bits 31—16) registers are multiplied and the result is placed in the p register. The bit alignment of the p register is a function of the ALIGN field of the auc register.
- **aD=aS+p p=x*y**. The contents of the source accumulator, aS, are added to the contents of the p register and the result is placed in the destination accumulator, aD. The bit alignment of the p register is a function of the ALIGN field of the auc register. The contents of the x and the y (bits 31—16) registers are multiplied and the result is placed in the p register.
- **aD=aS–p p=x*y**. The contents of the p register are subtracted from the contents of the source accumulator, aS, and the result is placed in the destination accumulator, aD. The bit alignment of the p register is a function of the ALIGN field of the auc register. The contents of the x and the y (bits 31—16) registers are multiplied and the result is placed in the p register.
- **aD=p**. The contents of the p register are copied into the destination accumulator, aD. The bit alignment of the p register is a function of the ALIGN field of the auc register.
- **aD=aS+p**. The contents of the source accumulator, aS, are added to the contents of the p register, and the result is placed in the destination accumulator, aD. The bit alignment of the p register is a function of the ALIGN field of the auc register.
- **aD=aS−p**. The contents of the p register are subtracted from the contents of the source accumulator, aS, and the result is placed in the destination accumulator, aD. The bit alignment of the p register is a function of the ALIGN field of the auc register.
- **aD=y**. The contents of the y register are copied into the destination accumulator, aD.
- **aD=aS+y**. The contents of the source accumulator, aS, are added to the contents of the y register and the result is placed in the destination accumulator, aD.
- **aD=aS−y**. The contents of the y register are subtracted from the contents of the source

accumulator, aS, and the result is placed in the destination accumulator, aD.

- **aD=aS&y**. The contents of the source accumulator, aS, are ANDed with the contents of the y register, and the result is placed in the destination accumulator, aD.
- **aD=aS|y**. The contents of the source accumulator, aS, are ORed with the contents of the y register, and the result is placed in the destination accumulator, aD.
- **aD=aS^y**. The contents of the source accumulator, aS, are XORed with the contents of the y register, and the result is placed in the destination accumulator, aD.
- **aS−y**. The contents of the y register are subtracted from the contents of the source accumulator, aS. The result is not placed in the destination accumulator, aD; however, the ALU flags are affected by the results of the subtraction.
- **aS&y**. The contents of the source accumulator, aS, are ANDed to the contents of the y register. The result is not placed in the destination accumulator, aD; however, the ALU flags are affected by the results of the AND function.

3.4.2 Transfer Statements

The transfer statements allow the user to transfer data from memory to the x and y registers and the accumulators, or from the y register and the accumulators to memory.

- **y=Y x=X**. The data from the specified Y source is loaded into the high half (bits 31—16) of the y register. The data from the specified X source is loaded into the x register. If clearing of yl is enabled (according to the CLR field of the auc register), then yl is cleared (0) when the high half is loaded.
- **y=aT x=X**. The data in the high half (bits 31—16) of the specified accumulator is loaded into the high half (bits 31—16) of the y register. The data from the specified X source is loaded into the x register. If clearing of yl is enabled (according to the CLR field of the auc register), then yl is cleared (0) when the high half is loaded.
- **y=Y**. The data from the specified Y source is loaded into the high half of the y register (bits 31—16). If clearing of yl is enabled (according to the CLR field of the auc register), then yl is cleared (0) when the high half is loaded.
- **yl=Y**. The data from the specified Y source is loaded into the low half of the y register (bits 15—0). The data in the high half of y is not altered.
- **aT=Y**. The data from the specified Y source is loaded into the high half (bits 31—16) of the specified accumulator. The guard bits (35—32) are loaded with the value of bit 31. If clearing of aTl is enabled (according to the CLR field of the auc register), the low half of the accumulator is cleared (0) when the high half is loaded.

- **aTl=Y**. The data from the specified Y source is loaded into the low half (bits 15—0) of the specified accumulator. The data in the high half of the accumulator is not altered.
- **x=Y**. The data from the specified Y source is loaded into the x register.
- **Y**. No data is transferred. This transfer statement is used to modify the address register specified.
- **Y=y**. The data in the high half of the y register (bits 31—16) is loaded into the specified Y destination.
- **Y=yl**. The data in the low half of the y register (bits 15—0) is loaded into the specified Y destination.
- **Y=aT**. The data in the high half (bits 31—16) of the specified accumulator is written into the specified Y destination. If saturation on overflow is selected (according to the SAT field of the auc register), the accumulator value is limited. See Section 2.5.1.
- **Y=aTl**. The data in the low half (bits 15—0) of the specified accumulator is written into the specified Y destination. If saturation on overflow is selected (according to the SAT field of the auc register), the accumulator value is limited. See Section 2.5.1.
- **Z:y x=X**. The data from the specified X source is loaded into the x register. The data from the specified Z source is loaded into the high half (bits 31—16) of the y register, and the old data from the high half of the y register is loaded into the Z destination. If clearing of yl is enabled (according to the CLR field of the auc register), then yl is cleared (0) when the high half is loaded.
- **Z:y**. The data from the specified Z source is loaded into the high half (bits 31—16) of the y register and the old data from the high half of the y register is loaded into the Z destination. If clearing of yl is enabled (according to the CLR field of the auc register), then yl is cleared (0) when the high half is loaded.
- **Z:yl**. The data from the specified Z source is loaded into the low half (bits 15—0) of the y register and the old data of the low half of the y register is loaded into the Z destination. Data in the high half of the y register is not altered.
- **Z:aT**. The data from the specified Z source is loaded into the high half (bits 31—16) of the specified accumulator. If clearing of aTl is enabled (according to the CLR field of the auc register), the low half of the accumulator is cleared (0) when the high half is loaded. The guard bits (35—32) are loaded with the value of bit 31. The old data from the high half of the accumulator is loaded into the Z destination. If saturation on overflow is enabled (according to the SAT field of the auc register), the accumulator value is limited. See Section 2.5.1.
- **Z:aTl**. The data from the specified Z source is loaded into the low half (bits 15—0) of the specified accumulator and the old data from the high half of the accumulator is loaded into the Z destination. The data in the high half of the accumulator is not altered. If saturation on overflow is enabled (according to the SAT field of the auc register), the accumulator value is limited. See Section 2.5.1.

3.4.3 No Operation

- **nop**. Single cycle no operation. **N * nop** (i.e., **4 * nop**) may be used to perform N no operation instructions.
- ;. The semicolon is an optional no operation mnemonic. **N *** ; may also be used to perform N no operation instructions.

3.5 SPECIAL FUNCTION GROUP

Instructions from the special function group are **always** executed in one instruction cycle. They require one word of program memory. Using the special function instructions, the DSP16/DSP16A device can be used to implement a number of algorithms, which include the following nonlinear functions: absolute value, signum, minimum and maximum value finder, A-law and μ-law conversions, division, half-wave and full-wave rectification, and rounding. Special function instructions are executed either conditionally or unconditionally. Both the condition and its complement are available for use in special function instructions. A special function instruction uses one ROM location. Instructions from this group can be used in the cache.

The special function instructions can be conditioned on the basis of the results of previous multiply/ALU and special function instructions, the value of one of the counters (c0, c1), or the value of a randomly set bit in the DSP16 device. The result of the most recent accumulator operation prior to the special function instruction establishes the state of the flags for the conditions associated with logical or mathematical functions.

The special functions given in Table 3-5 can be conditionally executed as **if** *CON instruction* and with an event counter as **ifc** *CON instruction*, meaning that:

```
if CON is true then
    c1 = c1 + 1
    instruction
    c2 = c1
else
    c1 = c1 + 1
```

Note: When using the event counter (**ifc** instruction), if *CON* is c0lt or c0gt, then c0 is not incremented; if *CON* is c1lt or c1gt, then c1 is incremented once.

Table 3-5. Special Function Instructions

Instruction	Description

aD=aS>>1
aD=aS>>4
aD=aS>>8
aD=aS>>16
	Arithmetic right shift (sign preserved) of 36-bit accumulators.

aD=aS
aD=–aS
	—

aD=rnd(aS)	Round upper 20 bits of accumulator.
aDh=aSh+1	Increment high half of accumulator (lower half cleared).
aD=aS+1	Increment accumulator.

aD=y
aD=p
	—

aD=aS<<1
aD=aS<<4
aD=aS<<8
aD=aS<<16
	Logical left shift (sign-extended from bit 31) of the least significant 32 bits of the 36-bit accumulators.

Table 3-6. Replacement Table for Special Function Instructions

Replace	Value	Meaning
aD,aS	a0,a1	One of two DAU accumulators.
CON	mi, pl, eq, ne, gt, le, lvs, mvs, mvc, c0ge, c0lt, c1ge, c1lt, heads, tails, true, false	See Table 3-2 for definitions of processor flags.

3.5.1 Special Function Statements

The statements must be written in the exact format shown. If the statements are written in any other way, for example, aD=1+aS instead of aD=aS+1, the assembler produces an error message.

- **aD=aS>>1.** The contents of the source accumulator, aS, are divided by 2 and the result is placed in the destination accumulator, aD. The sign bit is preserved.
- **aD=aS>>4.** The contents of the source accumulator, aS, are divided by 2^4 and the result is placed in the destination accumulator, aD. The sign bit is preserved.
- **aD=aS>>8.** The contents of the source accumulator, aS, are divided by 2^8 and the result is placed in the destination accumulator, aD. The sign bit is preserved.
- **aD=aS>>16.** The contents of the source accumulator, aS, are divided by 2^{16} and the result is placed in the destination accumulator, aD. The sign bit is preserved.
- **aD=aS<<1.** The contents of the source accumulator, aS, are logically shifted one bit left and the result is placed in the destination accumulator, aD. The sign bit is extended from bit 31.

- **aD=aS<<4.** The contents of the source accumulator, aS, are logically shifted four bits left and the result is placed in the destination accumulator, aD. The sign bit is extended from bit 31.
- **aD=aS<<8.** The contents of the source accumulator, aS, are logically shifted eight bits left and the result is placed in the destination accumulator, aD. The sign bit is extended from bit 31.
- **aD=aS<<16.** The contents of the source accumulator, aS, are logically shifted sixteen bits left and the result is placed in the destination accumulator, aD. The sign bit is extended from bit 31.
- **aD=aS.** The contents of the source accumulator, aS, are placed in the destination accumulator, aD.
- **aD=–aS.** The 2's complement of the contents of the source accumulator, aS, is placed in the destination accumulator, aD.
- **aD=rnd(aS).** The contents of the source accumulator, aS, are rounded to 16 bits, and the sign-extended result is placed in aD[35 — 16] with zeros in aD[15 — 0].
- **aDh=aSh+1.** The value 0x000010000 is added to the contents of the source accumulator, aS, and the result is placed in the destination accumulator, aD. This statement increments by one the data in the high half of the source accumulator. The low half of aD is cleared.
- **aD=aS+1.** The value 0x000000001 is added to the contents of the source accumulator, aS, and the result is placed in the destination accumulator, aD. This statement increments by one the data in the source accumulator.
- **aD=y.** The contents of the y register are written to the destination accumulator, aD.
- **aD=p.** The contents of the p register are written to the destination accumulator, aD. The bit alignment of the p register is a function of the ALIGN field of the auc register.

3.6 CONTROL GROUP

The control instructions allow the user to implement goto, call, and return commands. There is no latency when branching, i.e., the instruction executed following the control instruction has the address specified in the pc register after execution of the control instruction. Control instructions are executed either conditionally or unconditionally. Both the condition and its complement are available for use in control instructions. A control instruction uses one ROM location; conditional control instructions require two ROM locations. The execution time for an unconditional control instruction is two instruction cycles, and the execution time for conditional control instructions is three instruction cycles. The **icall** instruction executes in three cycles. Control instructions may not be executed in the cache.

The control instructions can be conditioned on the basis of the results of previous multiply/ALU and special function instructions, the value of one of the counters (c0, c1), or the value of a randomly set bit in the DSP16/DSP16A device. The result of the most recent accumulator operation prior to the control instruction establishes the state of the flags for the conditions associated with logical or mathematical functions.

An example of a control instruction conditionally executed is **if** *CON* **goto** *JA*.

Control Instructions*	
goto *JA*	icall†
goto pt	return (goto pr)
call *JA*	ireturn† (goto pi)
call pt	

* Control instructions cannot be used in the cache.
† **icall** and **ireturn** can not be conditionally executed.

Table 3-7. Replacement Table for Control Function Instructions

Replace	Value	Meaning
CON	mi, pl, eq, ne, gt, le, lvs, mvs, mvc, c0ge, c0lt, c1ge, c1lt, heads, tails, true, false	See Table 3-2 for definitions of processor flags.
JA	12-bit value	Least significant 12 bits of an absolute address within the same 4 Kword memory section.

3.6.1 Control Statements

- **goto JA**. The goto JA instruction moves the immediate value JA into the lower 12 bits of the program counter (pc) register, when goto JA is executed. The upper 4 bits of pc remain unchanged. The instruction with address JA is the next instruction executed. The goto JA instruction does not affect the program return (pr) register, and can be used in a subroutine without losing the return address of the subroutine.
- **call JA**. The call JA instruction moves the contents of the program counter (pc) register into the program return (pr) register and the immediate data JA into the lower 12 bits of the pc register. The upper 4 bits of pc remain unchanged. The pr register holds the return address of the subroutine (the address of the instruction following call JA); i.e., if call JA is located at address i, then the pr register is loaded with address i+1. The instruction with address N is the next instruction executed.
- **goto pt**. The goto pt instruction moves the contents of pt into the program counter (pc) register, when goto pt is executed. The instruction with address equal to the contents of pt is the next instruction executed. Since pt is a 16-bit register, goto pt allows branches to any location in the 64 Kword program space. The goto pt instruction does not affect the program return register.

- **call pt**. The call pt instruction moves the contents of the program counter (pc) register into the program return (pr) register and the data in pt into the pc register. The pr register holds the return address of the subroutine (the address of the instruction following call pt); i.e., if the call pt is located at address i, then the pr register is loaded with the value i+1. The instruction with address equal to the contents of pt is the next instruction executed.
- **icall.** The icall instruction moves the contents of the program counter (pc) register into the program interrupt (pi) register and the address 2 into the pc register. The pi register holds the return address of the interrupt routine (the address following the icall instruction); i.e., if the icall instruction is located at address i, then the pi register is loaded with the value i+1. The icall instruction is used by the DSP16/DSP16A Development Systems for breakpointing and is, therefore, reserved for that purpose when development system breakpoints are used.
- **return /goto pr**. The return instruction moves the contents of the program return (pr) register into the program counter (pc) register. The pr register holds the return address of the subroutine. Execution of the instruction with address equal to the contents of pr follows the execution of the return instruction. The goto pr instruction works identically to the return instruction.
- **ireturn /goto pi**. The ireturn instruction moves the contents of the program interrupt (pi) register into the program counter (pc) register. The pi register holds the interrupt return address. When an interrupt occurs, the value of the pc register is written into the pi register. Execution of the instruction with address equal to the contents of pi follows the execution of the ireturn instruction. The goto pi instruction works identically to the ireturn instruction.

3.7 DATA MOVE INSTRUCTIONS

Data move instructions transfer from a RAM location to a register, from a register to a RAM location, from an accumulator to a register, from a register to an accumulator; and load immediate data to a register. Data move instructions involving immediate data loaded into YAAU registers use one ROM location and execute in one instruction cycle if the data can be encoded in the instruction itself (R = M, M ≤ 9 bits) or two ROM locations if the data is not contained in the instruction (R = N). All other data move instructions use one ROM location. Data move instructions are executed in two instruction cycles except for those instructions in which the immediate data is encoded in the instruction which are executed in one instruction cycle as noted above (R = M). All data move instructions, with the exception of two-word immediate moves, may be executed inside the cache.

Data Move Instructions	
R = N	aT = R
R = M	Y = R
R = Y	Z : R
R = aS	

Table 3-8. Replacement Table for Data Move Instructions

Replace	Value	Meaning
R	x	DAU register – signed, 16 bits.
	y	DAU register – signed, 16 bits.[1]
	yl	DAU register – unsigned, 16 bits.
	auc	DAU control register – unsigned, 7 bits.
	c0	DAU counter 0 – signed, 8 bits.
	c1	DAU counter 1 – signed, 8 bits.
	c2	DAU counter 2 – signed, 8 bits.
	r0	YAAU ptr. reg. – unsigned, 9 bits (16 bits in DSP16A).
	r1	YAAU ptr. reg. – unsigned, 9 bits (16 bits in DSP16A).
	r2	YAAU ptr. reg. – unsigned, 9 bits (16 bits in DSP16A).
	r3	YAAU ptr. reg. – unsigned, 9 bits (16 bits in DSP16A).
	rb	YAAU mod. addr. reg. – unsigned, 9 bits (16 bits in DSP16A).
	re	YAAU mod. addr. reg. – unsigned, 9 bits (16 bits in DSP16A).
	j	YAAU inc. reg. – signed, 9 bits (16 bits in DSP16A).
	k	YAAU inc. reg. – signed, 9 bits (16 bits in DSP16A).
	pt	XAAU pointer register – unsigned, 16 bits.
	pr	XAAU program return register – unsigned, 16 bits.
	pi	XAAU program interrupt register – unsigned, 16 bits.[2]
	i	XAAU increment register – signed, 12 bits.
	psw	Processor status word.
	sioc	Serial I/O control register.[3]
	sdx	Serial I/O data register.
	tdms	Serial I/O tdms control register.[3]
	srta	Serial receive/transmit address.[3]
	pioc	Parallel I/O control register.
	pdx0	Parallel I/O data register with PSEL = 0 (pin 72).
	pdx1	Parallel I/O data register with PSEL = 1 (pin 72).
aD, aS	a0, a1	High half of accumulator.[1]
Y	*rM,*rM++, *rM−−,*rM++j	Same as in multiply/ALU instructions.
Z	*rMzp,*rMpz, *rMm2,*rMjk	Same as in multiply/ALU instructions.
N	16-bit value	Immediate data.
M	9-bit value	Immediate data for YAAU registers.

Notes:
When reading signed registers less than 16 bits wide, their contents are sign-extended to 16 bits. When reading unsigned registers less than 16 bits wide, their contents are zero-extended to 16 bits. When short immediate addressing is used to write to YAAU registers in the DSP16A, unsigned registers are zero-extended from 9 to 16 bits. Signed registers (j,k) are sign-extended from 9 to 16 bits.

[1]Data moves to y, a0, or a1 load the high half (bits 31—16) of the register. If clearing of the destination is enabled (according to the CLR field of the auc register), the low half of the destination register is cleared (0) when the high half is loaded.

[2]The pi register acts as a "shadow" of the pc register. Each time the pc changes, its value is also loaded into pi. "Shadowing" is disabled when executing an interrupt service routine, therefore, pi contains the contents of pc prior to the interrupt. Writes to pi do not alter its contents, except during interrupt service routines.

[3]sioc, tdms, and srta registers are not readable.

3.7.1 Data Move Instruction Statements

The data move instruction statements must be written in the exact format shown. If the statements are written in any other way, for example, R: Z instead of Z:R, the assembler generates incorrect code and produces an error message. Data move instructions execute in two instruction cycles and require 1 word of program memory (immediate loads, R = N, require two words of program memory). Short immediate data move instructions require one word of program memory and execute in one cycle.

- **R=N** loads the immediate data value, N, into the specified destination register, R. This form of the data move instructions may not be executed in the cache.
- **R=M** loads a 9-bit immediate data value, M, into one of the YAAU registers (rb, re, r0, r1, r2, or r3). This special case immediate instruction is often referred to as a "short immediate" or "register set" instruction. Short immediate instructions require one word of program memory, execute in one cycle, and may be executed inside the cache.
- **R=Y** loads the data contained in the specified Y source into the specified destination register, R.
- **R=aS** loads the data contained in bits 31—16 of the specified transfer accumulator, aS, into the specified destination register, R. If saturation on overflow is enabled (according to the SAT field of the auc register), then the accumulator is limited. (See Section 2.5.1.)
- **Y=R** loads the data contained in the specified source register, R, into the specified Y destination.
- **aT=R** loads the data contained in the specified source register, R, into bits 31—16 of the specified accumulator. If clearing of aTl is enabled (according to the CLR field of the auc register), then aTl is cleared (0) when the high half is loaded. The guard bits are loaded with the value of bit 31.
- **Z: R** writes data from the specified Z source to the specified R destination register and writes the old data in the source register, R, to the Z destination (see Section 3.2.4 for an explanation of this data transfer mode).

3.8 CACHE INSTRUCTIONS

The cache instructions allow the implementation of low overhead loops to conserve program memory. When used, the cache instruction treats the specified NI instructions as a loop to be executed K times. Both cache instructions use one ROM location. The **do** instruction executes in one instruction cycle, while the **redo** instruction executes in two instruction cycles.

Cache Instructions	

do *K* {
instruction1
instruction2
.
.
instructionNI
}
	redo *K*

Table 3-9. Replacement Table for Cache Instructions		
Replace	Value	Meaning
K	2 ≤ K ≤ 127	Number of times the instructions are to be executed.
NI	1 ≤ NI ≤ 15	1 to 15 instructions may be included.

3.8.1 Cache Statements

When the cache is used to repeat a block of NI instructions, the cycle timings of the instructions are as follows:

1. The "first pass" does not affect cycle timings except for the last instruction in the block of NI instructions. This instruction executes in two cycles.
2. During pass 2 through pass K+1, each instruction is executed "in the cache" (see Table 3-3).
3. During the last (Kth) pass, the block of instructions executes "inside the cache," except for the last instruction, which executes outside the cache.

The instructions remain in the cache memory and may be re-executed using the **redo** command without the need to reload the cache.

- **redo k.** When the redo k instruction is used, the DSP executes the NI instructions currently in the cache's memory k times. On the last iteration, the last instruction is executed outside the cache.

Note: Control group instructions and two-word data move instructions may not be executed from the cache.

3.9 INSTRUCTION SET SUMMARY

This section explains, in detail, the instruction set for the DSP16/DSP16A. Refer to Appendix A for instruction set formats and field encodings.

goto JA (branch direct)

(PC) ← (PC bits 15—12)(JA)

Program control jumps to location JA (within the same 4 Kword page). The lower 12 bits of the PC are written with the 12-bit immediate value of JA. The upper 4 bits of the PC remain unchanged (the **goto pt** instruction is used for branches outside the current 4 Kword page).

Bit	15			12	11 … 0
Field	0	0	0	0	JA

Words:	1
Cycles:	2
Group:	Control
Addressing:	Immediate
Flags affected:	None
Interruptible:	No
Cacheable:	No
Format:	4

goto B (branch direct)

(pc) ← (B)

Program control jumps to the location pointed to by the register encoded in the B field. The pc is written with the 16-bit value of the register. The following branch destinations are specified in the B field:

B Field	Action
000	return (same as goto pr)
001	ireturn (same as goto pi)
010	goto pt
011	call pt*
1xx	Reserved

* For this instruction, note that the current pc is also saved in the pr register before the jump.

Bit	15				11	10	8	7							0
Field	1	1	0	0	0	B		0	0	0	0	0	0	0	0

Words:	1
Cycles:	2
Group:	Control
Addressing:	Register
Flags affected:	None
Interruptible:	No
Cacheable:	No
Format:	5

if CON goto/call/return

(conditional branch qualifier)

test CONdition;
if true, execute the following control statement

The condition CON is tested (encoded in the CON field). If the condition is true, the next instruction (which must be a control instruction) is executed. If false, the control instruction is not executed. The CON field is encoded as:

CON	Flag	CON	Flag
00000	mi (negative result)	01001	tails (random bit clear)†
00001	pl (positive result)	01010	c0ge (counter0 ≥ 0)*
00010	eq (result = 0)	01011	c0lt (counter0 < 0)*
00011	ne (result ≠ 0)	01100	c1ge (counter1 ≥ 0)*
00100	lvs (logical overflow set)	01101	c1lt (counter1 < 0)*
00101	lvc (logical overflow clear)	01110	true (always)
00110	mvs (math. overflow set)	01111	false (never)
00111	mvc (math. overflow clear)	10000	gt (result > 0)
01000	heads (random bit set)†	10001	le (result ≤ 0)

* Using the c0ge or c0lt conditions also causes the value of the c0 counter to be postincremented.
Using the c1ge or c1lt conditions also causes the value of the c1 counter to be postincremented.
† The random bit is updated after each test of heads or tails.

The ensuing control opcode can be any of the following:

goto JA   goto pt   call JA   call pt   return (goto pr)

Note that **ireturn** and **icall** are the only control instructions that cannot be conditionally executed.

Bit		15										5	4 … 0
Field	word 1	1	1	0	1	0	0	0	0	0	0	0	CON
	word 2	CONTROL OPCODE											

Words:	1
Cycles:	3 (including the branch/call/return)
Group:	Control
Addressing:	None
Flags affected:	None
Interruptible:	No
Cacheable:	No
Format:	6

call JA (call subroutine direct)

(pr) ← (pc + 1)
(pc) ← (pc bits 15—12)(JA)

The subroutine at address JA (within the same 4 Kword page) is called. First the return address (the address of the first instruction following the call) is placed into the pr register. Then the lower 12 bits of the pc are written with the 12-bit immediate value of JA. The upper 4 bits of pc remain unchanged (the **call pt** instruction is used for calling subroutines out of the current 4 Kword page).

Bit	15			12	11 ... 0
Field	1	0	0	0	JA

Words:	1
Cycles:	2
Group:	Control
Addressing:	Immediate
Flags affected:	None
Interruptible:	No
Cacheable:	No
Format:	4

icall (software interrupt)

(pi) ← (pc + 1)
(pc) ← 2
IACK

The interrupt handler is called, just as it would be by an external interrupt. The interrupt return register is set to next pc + 1, and the pc is set to 2, to start execution at the interrupt handler. Note that external interrupts vector to location 1, and icall vectors to location 2. The interrupt acknowledge pin (IACK) is set just as it would be by an external interrupt.

Bit	15															0
Field	1	1	0	1	0	1	0	0	0	0	0	0	1	1	1	0

Words:	1
Cycles:	3
Group:	Control
Addressing:	None
Flags affected:	None
Interruptible:	No
Cacheable:	No
Format:	6

do K {

```
do K {
    instr1
      •
      •          (loop in cache; cache loaded with new contents)
      •
    instrNI
    }
```

execute the next NI instructions K times

The next NI instructions are loaded into the cache concurrent with their execution. They are then executed within the cache K–1 more times, at (potentially) higher speed.

The iteration count K can be between 2 and 127, inclusive, and the number of instructions NI must be between 1 and 15, inclusive.

Notes on cache performance:

The do instruction executes in one cycle. When the cache is used to repeat a block of NI instructions, the cycle timings of the instructions are as follows:

1. The "first pass" does not affect cycle timings except for the last instruction in the block of NI instructions. This instruction executes in two cycles.
2. During pass 2 through pass K+1, each instruction is executed "in the cache" (see Table 3-3).
3. During the last (Kth) pass, the block of instructions executes "inside the cache" except for the last instruction, which executes outside the cache.

The instructions remain in the cache memory and may be re-executed using the **redo** command without the need to reload the cache.

Bit	15				11	10 7	6 0
Field	0	1	1	1	0	NI	K

Words:	1
Cycles:	1
Group:	Cache
Addressing:	Immediate
Flags affected:	None
Interruptible:	No
Cacheable:	No
Format:	10

redo K (loop in cache; cache contents unaffected)

execute the current contents of the cache K times

The current contents of the cache (loaded with a previous **do** instruction) are executed within the cache K additional times. The iteration count K can be between 2 and 127, inclusive.

Notes on cache performance:

The **redo** instruction executes in two cycles. All instructions require the in-cache time to execute, except the last instruction of the last iteration, which requires the out-of-cache time to execute. Thereafter, instructions (fetched from ROM) require their normal out-of-cache time to execute.

Bit	15				11	10			7	6	0
Field	0	1	1	1	0	0	0	0	0	K	

Words:	1
Cycles:	2
Group:	Cache
Addressing:	Immediate
Flags affected:	None
Interruptible:	No
Cacheable:	No
Format:	10

R = M (short immediate load)

(R) ← (M)

The contents of register R are replaced with the 9-bit immediate value of M. The value of R can be any of the following:

Register	R	Register	R
j	000	r0	100
k	001	r1	101
rb	010	r2	110
re	011	r3	111

For the DSP16, these registers are all 9 bits wide. For the DSP16A, these registers are 16 bits wide and the j and k registers are sign-extended (2's complement). The others are zero-extended.

Bit	15			12	11	9	8	0
Field	0	0	0	1	R		M	

Words:	1
Cycles:	1
Group:	Data Move
Addressing:	Immediate
Flags affected:	None
Interruptible:	Yes
Cacheable:	Yes
Format:	9

Notes:

1) In Appendix A, this instruction is encoded using a 2-bit I field that corresponds to the two LSBs of the R field shown above. The most significant bit of R is the least significant bit of the T field used in the instruction set encodings in Appendix A.

2) When a DSP16A program is encoded, if the immediate value M is greater than 9 bits or if a label is used for M, the assembler defaults to a two-word, two-cycle data move encoding. The short immediate encoding can be forced by using the optional mnemonic *set* (if the value of M is greater than 9 bits, it is truncated to 9 bits). For example:

   set r3 = var1

   forces a short immediate encoding.

R = N (16-bit immediate load)

(R) ← (N)

The contents of register R are replaced with the 16-bit immediate value of N. The value of R can be any of the following:

Register	R Field	Register	R Field
r0 (u)	000000	yl	010010
r1 (u)	000001	auc (u)	010011
r2 (u)	000010	psw	010100
r3 (u)	000011	c0 (s)	010101
j (s)	000100	c1 (s)	010110
k (s)	000101	c2 (s)	010111
rb (u)	000110	sioc	011000
re (u)	000111	srta	011001
pt	001000	sdx	011010
pr	001001	tdms	011011
pi	001010	pioc	011100
i (s)	001011	pdx0	011101
x	010000	pdx1	011110
y	010001		

Register sources j, k, i, c0, c1, and c2 are less than 16 bits and are sign-extended (s). Register sources r0, r1, r2, r3, rb, re, and auc are less than 16 bits and are zero-extended (u). For the DSP16A, registers r0, r1, r2, r3, j, k, rb, and re are 16 bits wide and need no sign- or zero-extension.

Note: writing the psw also writes the a0 and a1 guard bits.

Bit		15					10	9 … 4	3			0
Field	word 1	0	1	0	1	0	0	R	0	0	0	0
	word 2	Immediate Value (N)										

Words:	2
Cycles:	2
Group:	Data Move
Addressing:	Immediate
Flags affected:	None
Interruptible:	Yes
Cacheable:	No
Format:	8

R = aS (load register from accumulator)

(R) ← (aS)

The contents of register R are replaced with the current contents of bits 31—16 of accumulator aS. Registers which are less than 16 bits load from the low-order bits of aS[31—16].

The value of S can be 0 to select accumulator a0 or 1 to select accumulator a1. See Appendix A for the possible values of R.
Note: Writing the psw also writes the the a0 and a1 guard bits.

Bit	15					10	11 — 4	3			0
Field	0	1	0	S	1	0	R	0	0	0	0

Words:	1
Cycles:	2
Group:	Data Move
Addressing:	Register
Flags affected:	None
Interruptible:	Yes
Cacheable:	Yes
Format:	7

aT = R (load accumulator from register)

(aT) ← (R)

The contents of bits 31—16 of accumulator aT are replaced with the current contents of register R, zero- or sign-extended to 16 bits (if necessary). If clearing aTl is enabled (with the CLR field of the auc register), bits 15—0 of accumulator aT will be cleared. Bits 35—32 (the guard bits) will be loaded with copies of bit 31.

The value of $\overline{aT}$ can be 0 to select a1, or 1 to select a0. (aT is encoded as $\overline{aT}$ in the instruction encodings in Appendix A.) The value of R can be any of the following:

Register	R Field	Register	R Field
r0 (u)	000000	yl	010010
r1 (u)	000001	auc (u)	010011
r2 (u)	000010	psw	010100
r3 (u)	000011	c0 (s)	010101
j (s)	000100	c1 (s)	010110
k (s)	000101	c2 (s)	010111
rb (u)	000110	sioc	011000
re (u)	000111	srta	011001
pt	001000	sdx	011010
pr	001001	tdms	011011
pi	001010	pioc	011100
i (s)	001011	pdx0	011101
x	010000	pdx1	011110
y	010001		

Register sources j, k, i, c0, c1, and c2 are less than 16 bits and are sign-extended (s). Register sources r0, r1, r2, r3, rb, re, and auc are less than 16 bits and are zero-extended (u). For the DSP16A, registers r0, r1, r2, r3, j, k, rb, and re are 16 bits wide and need no sign- or zero-extension.

Bit	15				11	10	9 4	3			0
Field	0	1	0	0	0	$\overline{a}$T	R	0	0	0	0

Words:	1
Cycles:	2
Group:	Data Move
Addressing:	Register
Flags affected:	None
Interruptible:	Yes
Cacheable:	Yes
Format:	7a

Note: If y is used as the register R, the assembler forces a special function encoding. The resulting instruction moves all 32 bits (sign extended to 36 bits) of y into aT. All DAU flags are affected, and the execution requires only one cycle. If a two-cycle data move is desired, the optional mnemonic *move* may be used. Only the upper 16 bits of y are transferred and no flags are affected. Example:

```
move a0 = y
```

R = Y (load register from internal RAM)

perform (R) ← (*rN); then
modify rN

The contents of register R are replaced with the current contents of the internal RAM location pointed to by rN, where rN is specified by the two most significant bits of the Y field.

00 - r0 01 - r1 10 - r2 11 - r3

The value of rN is then postmodified, where the postmodification is specified by the two least significant bits of the Y field.

2 LSBs of Y	Action	Symbol
00	no action	*rN
01	postincrement	*rN++
10	postdecrement	*rN--
11	postincrement by (j)	*rN++j

Code 11, in this case, means add the current value of the j register to rN (after accessing *rN).

See Appendix A for the possible values of destination register R. Registers which are less than 16 bits load from the low-order bits of the memory location. Note: writing the psw also writes the a0 and a1 guard bits.

Bit	15					10	9 … 4	3 … 0
Field	0	1	1	1	1	0	R	Y

Words:	1
Cycles:	2
Group:	Data Move
Addressing:	Register, Register Indirect
Flags affected:	None
Interruptible:	Yes
Cacheable:	Yes
Format:	7

Note: If y, yl, or x is the destination register, R, the assembler assembles this instruction as a single-cycle multiply/ALU instruction. If a two-cycle move encoding is necessary, the optional mnemonic *move* may be used. For example:

```
move y = *r1
```

forces a move encoding.

Y = R (store register to RAM memory)

(*rN) ← (R); then
modify rN

The contents of the RAM memory location pointed to by rN are replaced with the current contents of register R, zero- or sign-extended to 16 bits (if necessary). rN is specified the two most significant bits of the Y field:

00 - r0   01 - r1   10 - r2   11 - r3

The value of rN is then postmodified, where the postmodification is specified by the two least significant bits of the Y field.

2 LSBs of Y	Action	Symbol
00	no action	*rN
01	postincrement	*rN++
10	postdecrement	*rN--
11	postincrement by (j)	*rN++j

Code 11, in this case, means add the current value of the j register to rN (after accessing *rN).

See Appendix A for possible values of R. Register sources j, k, i, c0, c1, and c2 are less than 16 bits and are sign-extended. Register sources r0, r1, r2, r3, rb, re, and auc are less than 16 bits and are zero-extended. For the DSP16A, registers r0, r1, r2, r3, j, k, rb, and re are 16 bits and need no sign- or zero-extending.

Bit	15				11	10	9 … 4	3 … 0
Field	0	1	1	0	0	x	R	Y

Words:	1
Cycles:	2
Group:	Data Move
Addressing:	Register, Register Indirect
Flags affected:	None
Interruptible:	Yes
Cacheable:	Yes
Format:	7

Z : R (exchange register with RAM memory)

temp ← (R); then
(R) ← (*rN); then
modify rN (first action); then
(*rN) ← temp; then
modify rN (second action)

The contents of the RAM memory location(s) pointed to by rN are exchanged with the current contents of register R, which is sign- or zero-extended to 16 bits (if necessary). The pointer rN is modified after each of the two memory accesses according to the M field. rN is specified by the two most significant bits of the Z field:

00 - r0	01 - r1	10 - r2	11 - r3

The available options for the postmodification are specified by the two least significant bits of the Z field as follows:

Symbol	2 LSBs of Z	First Action	Second Action
*rNzp	00	no action	postincrement
*rNpz	01	postincrement	no action
*rNm2	10	postdecrement	postincrement by 2
*rNjk	11	postincrement by (j)	postincrement by (k)

Code 11, in this case, means add the current value of the j register to rN after reading *rN, then add the current value of the k register to rN after writing *rN.

See Appendix A for possible values of R. Register sources j, k, i, c0, c1, and c2 are less than 16 bits and are sign-extended. Register sources r0, r1, r2, r3, rb, re, and auc are less than 16 bits and are zero-extended. For the DSP16A, registers r0, r1, r2, r3, j, k, rb, and re are 16 bits and need no sign- or zero-extension. Note: writing the psw also writes the a0 and a1 guard bits.

Bit	15				11	10	9 4	3 0
Field	0	1	1	0	1	x	R	Z

Words:	1
Cycles:	2
Group:	Data Move
Addressing:	Register, Register Indirect
Flags affected:	None
Interruptible:	Yes
Cacheable:	Yes
Format:	7

Note: R and rM must not be the same register (i.e., r2pz:r2). The two logical PIO registers,pdx0 and pdx1, cannot be used in compound data moves.

if CON F2

(If CONdition is true, then perform special function instruction)

test CONdition;
if true, then perform F2

The specified condition is tested. If it is true, the special function operation F2 is performed. See Appendix A for the conditions that can be tested (encoded in the CON field).

The F2 functions (special function group) that can be conditionally performed (encoded in the F2 field) are as follows:

F2	Operation
0000	aD = aS >> 1
0001	aD = aS << 1
0010	aD = aS >> 4
0011	aD = aS << 4
0100	aD = aS >> 8
0101	aD = aS << 8
0110	aD = aS >> 16
0111	aD = aS << 16
1000	aD = p
1001	aDh = aSh + 1
1010	Reserved
1011	aD = rnd(aS)
1100	aD = y
1101	aD = aS + 1
1110	aD = aS
1111	aD = −aS

Bit	15				11	10	9	8 5	4 0
Field	1	0	0	1	1	D	S	F2	CON

Words:	1
Cycles:	1
Group:	Special Function
Addressing:	Register
Flags affected:	All
Interruptible:	Yes
Cacheable:	Yes
Format:	3

ifc CON F2

(if CONdition is true, then perform special function instruction)
(modify counter1,2 accordingly)

counter c1 = c1 + 1;
test CONdition; if true then {perform F2; c2 = c1}

First, counter c1 is incremented. Next, the specified condition is tested. If the condition is true, the special function operation F2 is performed and counter c2 is set to the value of c1. The conditions that can be tested are encoded in the CON field (see Appendix A).

The possible F2 special functions that can be conditionally performed are:

F2	Operation
0000	aD = aS >> 1
0001	aD = aS << 1
0010	aD = aS >> 4
0011	aD = aS << 4
0100	aD = aS >> 8
0101	aD = aS << 8
0110	aD = aS >> 16
0111	aD = aS << 16
1000	aD = p
1001	aDh = aSh + 1
1010	Reserved
1011	aD = rnd(aS)
1100	aD = y
1101	aD = aS + 1
1110	aD = aS
1111	aD = –aS

The D and S fields are used to specify aD and aS.

Bit	15 … 11	10	9	8 … 5	4 … 0
Field	1 0 0 1 0	D	S	F2	CON

Words:	1
Cycles:	1
Group:	Special Function
Addressing:	Register
Flags affected:	All
Interruptible:	Yes
Cacheable:	Yes
Format:	3

F1 Y (multiply/ALU operation with postmodification of pointer register)

perform operation F1; then
access *rN; then
postmodify rN (the contents of *rN are not written to a destination)

This instruction performs the following three operations (effectively in sequence):

1. The operation F1 is performed. The possible F1 operations are:

F1	Operation	
0000	aD = p	p = x*y
0001	aD = aS + p	p = x*y
0010		p = x*y
0011	aD = aS – p	p = x*y
0100	aD = p	
0101	aD = aS + p	
0110	NOP	
0111	aD = aS – p	
1000	aD = aS | y	
1001	aD = aS ^ y	
1010	aS & y	
1011	aS – y	
1100	aD = y	
1101	aD = aS + y	
1110	aD = aS & y	
1111	aD = aS – y	

The value of S can be 0 to select a0 or 1 to select a1. The value of D can be 0 to select a0 or 1 to select a1. Flags are modified based on the value computed by the DAU. Note: for all diadic operations involving the y register, y is sign-extended to 36 bits before performing the operation (this includes logical operations).

2. Access the internal RAM location pointed to by rN, where rN is specified by the two most significant bits of the Y field as follows (the accessed location is not written to a destination):

   00 - r0 01 - r1 10 - r2 11 - r3

3. Postmodify the value of rN, where the postmodification is specified by the two least significant bits of the Y field.

2 LSBs of Y	Action	Symbol
00	no action	*rN
01	postincrement	*rN++
10	postdecrement	*rN−−
11	postincrement by (j)	*rN++j

   Code 11, in this case, means add the current value of the j register to rN (after accessing *rN).

Bit	15				11	10	9	8 5	4	3 0
Field	0	0	1	1	0	D	S	F1	0	Y

Words:	1
Cycles:	1
Group:	Multiply/ALU
Addressing:	Register Indirect, Register
Flags affected:	All
Interruptible:	Yes
Cacheable:	Yes
Format:	1

F1 Y = a0[l] (multiply/ALU operation with parallel accumulator store)
F1 Y = a1[l]

write the value of aT[l] to *rN; then
modify rN; then
perform operation F1

This instruction performs the following three operations (effectively in sequence):

1. Write the (old) value of a0, a1, a0l, or a1l to the internal RAM location pointed to by rN, where rN is specified by the two most significant bits of the Y field.

   00 - r0 01 - r1 10 - r2 11 - r3

   The X field selects y or yl:

   $X = 0 \rightarrow$ yl $X = 1 \rightarrow$ y

2. Postmodify the value of rN, where the postmodification is specified by the two least significant bits of the Y field.

2 LSBs of Y	Action	Symbol
00	no action	*rN
01	postincrement	*rN++
10	postdecrement	*rN−−
11	postincrement by (j)	*rN++j

   Code 11 in this case means add the current value of the j register to rN (after accessing *rN).

3. The operation F1 is performed. The possible operations for F1 are:

F1	Operation	
0000	aD = p	p = x*y
0001	aD = aS + p	p = x*y
0010		p = x*y
0011	aD = aS – p	p = x*y
0100	aD = p	
0101	aD = aS + p	
0110	NOP	
0111	aD = aS – p	
1000	aD = aS I y	
1001	aD = aS ^ y	
1010	aS & y	
1011	aS – y	
1100	aD = y	
1101	aD = aS + y	
1110	aD = aS & y	
1111	aD = aS – y	

The value of S can be 0 to select a0 or 1 to select a1. The value of D can be 0 to select a0 or 1 to select a1. Note: for all diadic operations involving the y register, y is sign-extended to 36 bits before performing the operation (this includes logical operations).

Bit		15				11	10	9	8 5	4	3 0
Field	a0	1	1	1	0	0	D	S	F1	X	Y
	a1	0	0	1	0	0	D	S	F1	X	Y

Words:	1
Cycles:	2
Group:	Multiply/ALU
Addressing:	Register Indirect, Register
Flags affected:	All
Interruptible:	Yes
Cacheable:	Yes
Format:	1

F1 x = Y (multiply/ALU operation with parallel load of x register)

perform operation F1; then
copy *rN to x; then
modify rN

This instruction performs the following three operations (effectively in sequence):

1. The multiply/ALU operation F1 is performed. The possible operations for F1 are as follows:

F1	Operation	
0000	aD = p	p = x*y
0001	aD = aS + p	p = x*y
0010		p = x*y
0011	aD = aS – p	p = x*y
0100	aD = p	
0101	aD = aS + p	
0110	NOP	
0111	aD = aS – p	
1000	aD = aS I y	
1001	aD = aS ^ y	
1010	aS & y	
1011	aS – y	
1100	aD = y	
1101	aD = aS + y	
1110	aD = aS & y	
1111	aD = aS – y	

The value of S can be 0 to select a0 or 1 to select a1. The value of D can be 0 to select a0 or 1 to select a1. Flags are modified based on the value computed by the DAU. Note: for all diadic operations involving the y register, y is sign-extended to 36 bits before performing the operation (this includes logical operations).

2. Access the internal RAM location pointed to by rN, and write this value into the x register. rN is specified by the most significant bits of the Y field:

   00 - r0   01 - r1   10 - r2   11 - r3

3. Postmodify the value of rN, where the postmodification is specified by the two least significant bits of the Y field.

2 LSBs of Y	**Action**	**Symbol**
00	no action	*rN
01	postincrement	*rN++
10	postdecrement	*rN– –
11	postincrement by (j)	*rN++j

Code 11, in this case, means add the current value of the j register to rN (after accessing *rN).

Bit	15				11	10	9	8 5	4	3 0
Field	1	0	1	1	0	D	S	F1	0	Y

Words:	1
Cycles:	1
Group:	Multiply/ALU
Addressing:	Register Indirect, Register
Flags affected:	All
Interruptible:	Yes
Cacheable:	Yes
Format:	1

F1 y[l] = Y (multiply/ALU operation with parallel load of y register)

perform operation F1; then
copy *rN to y (or yl); then
modify rN

This instruction performs the following three operations (effectively in sequence):

1. The multiply/ALU operation F1 is performed. The possible F1 operations are as follows:

F1	Operation	
0000	aD = p	p = x*y
0001	aD = aS + p	p = x*y
0010		p = x*y
0011	aD = aS – p	p = x*y
0100	aD = p	
0101	aD = aS + p	
0110	NOP	
0111	aD = aS – p	
1000	aD = aS | y	
1001	aD = aS ^ y	
1010	aS & y	
1011	aS – y	
1100	aD = y	
1101	aD = aS + y	
1110	aD = aS & y	
1111	aD = aS – y	

The value of S can be 0 to select a0 or 1 to select a1. The value of D can be 0 to select a0 or 1 to select a1. Flags are modified based on the value computed by the DAU. Note: for all diadic operations involving the y register, y is sign-extended to 36 bits before performing the operation (this includes logical operations).

2. Access the internal RAM location pointed to by rN, and write this value into the y (or yl) register. rN is specified by the two most significant bits of the Y field:

00 - r0	01 - r1	10 - r2	11 - r3

The X field selects y or yl:

$X = 0 \rightarrow yl \qquad X = 1 \rightarrow y$

3. Postmodify the value of rN, where the postmodification is specified by the two least significant bits of the Y field:

2 LSBs of Y	**Action**	**Symbol**
00	no action	*rN
01	postincrement	*rN++
10	postdecrement	*rN--
11	postincrement by (j)	*rN++j

Code 11, in this case, means add the current value of the j register to rN (after accessing *rN).

Bit	15				11	10	9	8 5	4	3 0
Field	1	0	1	1	1	D	S	F1	X	Y

Words:	1
Cycles:	1
Group:	Multiply/ALU
Addressing:	Register Indirect, Register
Flags affected:	All
Interruptible:	Yes
Cacheable:	Yes
Format:	1

F1 y = Y x = *pt++[i]

(multiply/ALU operation with parallel load of x and y registers)

perform operation F1; then
(y) ← (*rN); then
modify rN; then
(x) ← (*pt); then
(pt)= (pt)+ [1 or i]

This instruction performs the following operations (effectively in sequence):

1. The operation F1 is performed. The possible operations for F1 are:

F1	Operation	
0000	aD = p	p = x*y
0001	aD = aS + p	p = x*y
0010		p = x*y
0011	aD = aS – p	p = x*y
0100	aD = p	
0101	aD = aS + p	
0110	NOP	
0111	aD = aS – p	
1000	aD = aS | y	
1001	aD = aS ^ y	
1010	aS & y	
1011	aS – y	
1100	aD = y	
1101	aD = aS + y	
1110	aD = aS & y	
1111	aD = aS – y	

The value of S can be 0 to select a0 or 1 to select a1. The value of D can be 0 to select a0 or 1 to select a1. Flags are modified based on the value computed by the DAU. Note: for all diadic operations involving the y register, y is sign-extended to 36 bits before performing the operation (this includes logical operations).

2. Access the internal RAM location pointed to by rN, and write this value into the y register. rN is specified by the two most significant bits of the Y field:

   00 - r0 01 - r1 10 - r2 11 - r3

3. Postmodify the value of rN, where the postmodification is specified by the two least significant bits of the Y field:

2 LSBs of Y	Action	Symbol
00	no action	*rN
01	postincrement	*rN++
10	postdecrement	*rN−−
11	postincrement by (j)	*rN++j

   Code 11, in this case, means add the current value of the j register to rN (after accessing *rN).

4. Access the ROM location pointed to by pt, and write this value into the x register. Either internal or external ROM may be accessed, depending on the state of the EXM pin (and the address, in the case of the DSP16A).

5. Postmodify the value of the pt register by either 1 or i, selected by the X field:

   X = 0 → *pt++ X = 1 → *pt++i

Bit	15 11	10	9	8 5	4	3 0
Field	1 1 1 1 1	D	S	F1	X	Y

Words:	1
Cycles:	2 (1 cycle if in cache)
Group:	Multiply/ALU
Addressing:	Register Indirect, Register
Flags affected:	All
Interruptible:	Yes
Cacheable:	Yes
Format:	1

F1 y = a0 x = *pt++[i]
F1 y = a1 x = *pt++[i]

(multiply/ALU operation with parallel load of x and y registers)

perform operation F1; then
(y) ← (a0) or (a1); then
(x) ← (*pt); then
(pt)= (pt)+ [1 or i]

This instruction performs the following operations (effectively in sequence):

1. The operation F1 is performed. The possible operations for F1 are:

F1	Operation	
0000	aD = p	p = x*y
0001	aD = aS + p	p = x*y
0010		p = x*y
0011	aD = aS – p	p = x*y
0100	aD = p	
0101	aD = aS + p	
0110	NOP	
0111	aD = aS – p	
1000	aD = aS | y	
1001	aD = aS ^ y	
1010	aS & y	
1011	aS – y	
1100	aD = y	
1101	aD = aS + y	
1110	aD = aS & y	
1111	aD = aS – y	

The value of S can be 0 to select a0 or 1 to select a1. The value of D can be 0 to select a0 or 1 to select a1. Flags are modified based on the value computed by the DAU. Note: for all diadic operations involving the y register, y is sign-extended to 36 bits before performing the operation (this includes logical operations).

2. Copy the value in a0 or a1 to the y register. Note that the value copied from a0 or a1 is the value before executing the F1 operation, due to pipelining.
3. Access the ROM location pointed to by pt, and write this value into the x register. Either internal or external ROM may be accessed, depending on the state of the EXM pin (and the address, for the DSP16A).
4. Postmodify the value of the pt register by either 1 or i, selected by the X field:

   X = 0 $\rightarrow$ *pt++ X = 1 $\rightarrow$ *pt++i

Bit		15				11	10	9	8	5	4	3			0
Field	a0	1	1	0	0	1	D	S	F1		X	0	0	0	0
	a1	1	1	0	1	1	D	S	F1		X	0	0	0	0

Words:	1
Cycles:	2 (1 cycle if in cache)
Group:	Multiply/ALU
Addressing:	Register Indirect, Register
Flags affected:	All
Interruptible:	Yes
Cacheable:	Yes
Format:	1

F1 aT[l] = Y (multiply/ALU operation with parallel load of accumulator register)

perform operation F1; then
copy *rN to aT (or aTl); then
modify rN by M

This instruction performs the following three operations (effectively in sequence):

1. The operation F1 is performed. The possible operations for F1 are:

F1	Operation	
0000	aD = p	p = x*y
0001	aD = aS + p	p = x*y
0010		p = x*y
0011	aD = aS − p	p = x*y
0100	aD = p	
0101	aD = aS + p	
0110	NOP	
0111	aD = aS − p	
1000	aD = aS | y	
1001	aD = aS ^ y	
1010	aS & y	
1011	aS − y	
1100	aD = y	
1101	aD = aS + y	
1110	aD = aS & y	
1111	aD = aS − y	

The value of S can be 0 to select a0 or 1 to select a1. The value of $\overline{\text{aT}}$ can be 0 to select a1 or 1 to select a0. Since aD and aT must be different accumulators, aD will be the opposite of aT. Flags are modified based on the value computed by the DAU. Note: for all diadic operations involving the y register, y is sign-extended to 36 bits before performing the operation (this includes logical operations).

2. Access the internal RAM location pointed to by rN, and write this value to the aT (or aTl) register. $\overline{aT}$ is defined as the opposite of D for this instruction. Therefore, if the F1 field selects writing to aD, aD will be the opposite of aT. rN is specified by the two most significant bits of the Y field:

   00 - r0 01 - r1 10 - r2 11 - r3

   The X field selects y or yl:

   $X = 0 \rightarrow yl$ $X = 1 \rightarrow y$

3. Postmodify the value of rN, where the postmodification is specified the two least significant bits of the Y field:

2 LSBs of Y	**Action**	**Symbol**
00	no action	*rN
01	postincrement	*rN++
10	postdecrement	*rN--
11	postincrement by (j)	*rN++j

   Code 11, in this case, means add the current value of the j register to rN (after accessing *rN).

Bit	15				11	10	9	8 5	4	3 0
Field	1	0	1	1	1	$\overline{aT}$	S	F1	X	Y

Words:	1
Cycles:	1
Group:	Multiply/ALU
Addressing:	Register Indirect, Register
Flags affected:	All
Interruptible:	Yes
Cacheable:	Yes
Format:	1a

F1 Y = y[l] (multiply/ALU operation with parallel store of y register)

perform operation F1;
(*rN) ← (y) or (yl); then
modify rN

This instruction performs the following operations (effectively in sequence):

1. The operation F1 is performed. The possible operations for F1 are:

F1	Operation	
0000	aD = p	p = x*y
0001	aD = aS + p	p = x*y
0010		p = x*y
0011	aD = aS − p	p = x*y
0100	aD = p	
0101	aD = aS + p	
0110	NOP	
0111	aD = aS − p	
1000	aD = aS | y	
1001	aD = aS ^ y	
1010	aS & y	
1011	aS − y	
1100	aD = y	
1101	aD = aS + y	
1110	aD = aS & y	
1111	aD = aS − y	

The value of S can be 0 to select a0 or 1 to select a1. The value of D can be 0 to select a0 or 1 to select a1. Flags are modified based on the value computed by the DAU. Note: for all diadic operations involving the y register, y is sign-extended to 36 bits before performing the operation (this includes logical operations).

2. Write the value of y or yl to the internal RAM location pointed to by rN, where N is specified by the two most significant bits of the Y field:

   00 - r0 01 - r1 10 - r2 11 - r3

   The X field selects y or yl:

   X = 0 → yl X = 1 → y

3. Postmodify the value of rN, where the postmodification is specified by the two least significant bits of the Y field:

2 LSBs of Y	**Action**	**Symbol**
00	no action	*rN
01	postincrement	*rN++
10	postdecrement	*rN−−
11	postincrement by (j)	*rN++j

   Code 11, in this case, means add the current value of the j register to rN (after accessing *rN).

Bit	15				11	10	9	8 5	4	3 0
Field	1	0	1	0	0	D	S	F1	X	Y

Words:	1
Cycles:	2
Group:	Multiply/ALU
Addressing:	Register Indirect, Register
Flags affected:	All
Interruptible:	Yes
Cacheable:	Yes
Format:	1

F1 Z : y[l] (multiply/ALU operation with compound data move)

perform operation F1; then
temp ← (y) or (yl); then
(y) or (yl) ← (*rN); then
modify rN (first action); then
(*rN) ← temp; then
modify rN (second action)

This instruction performs the following operations (effectively in sequence):

1. The operation F1 is performed. The possible F1 operations are:

F1	Operation	
0000	aD = p	p = x*y
0001	aD = aS + p	p = x*y
0010		p = x*y
0011	aD = aS – p	p = x*y
0100	aD = p	
0101	aD = aS + p	
0110	NOP	
0111	aD = aS – p	
1000	aD = aS | y	
1001	aD = aS ^ y	
1010	aS & y	
1011	aS – y	
1100	aD = y	
1101	aD = aS + y	
1110	aD = aS & y	
1111	aD = aS – y	

The value of S can be 0 to select a0 or 1 to select a1. The value of D can be 0 to select a0 or 1 to select a1. Flags are modified based on the value computed by the DAU. Note: for all diadic operations involving the y register, y is sign-extended to 36 bits before performing the operation (this includes logical operations).

2. Save either the y or yl register into an internal temporary location (temp). The X field select y or yl:

   X = 0 → yl   X = 1 → y

3. Access the internal RAM location pointed to by rN, and write this value into the y (or yl) register. rN is specified by the 2 most significant bits of the Z field:

   00 - r0   01 - r1   10 - r2   11 - r3

4. Postmodify the value of rN by the first action described by the two least significant bits of the Z field (described below).
5. Write the value saved in the temporary register (temp) to the memory location now pointed to by rN.
6. Postmodify the value of rN by the second action described by the two least significant bits of the Z field. The available options for the postmodification are specified as follows:

Symbol	2 LSBs of Z	First Action	Second Action
*rNzp	00	no action	postincrement
*rNpz	01	postincrement	no action
*rNm2	10	postdecrement	postincrement by 2
*rNjk	11	postincrement by (j)	postincrement by (k)

Code 11, in this case, means add the current value of the j (or) k register to rN (after accessing *rN).

Bit	15 ... 11	10	9	8 ... 5	4	3 ... 0
Field	1 0 1 0 1	D	S	F1	X	Z

Words:	1
Cycles:	2
Group:	Multiply/ALU
Addressing:	Register Indirect, Register
Flags affected:	All
Interruptible:	Yes
Cacheable:	Yes
Format:	2

F1 Z : aT[l] (multiply/ALU operation with parallel compound accumlator move)

```
perform operation F1; then
temp ← (aT) or (aTl); then
(aT) or (aTl) ← (*rN); then
modify rN (first action);
(*rN) ← temp;
modify rN (second action)
```

This instruction performs the following operations (effectively in sequence):

1. The operation F1 is performed. The possible operations for F1 are:

F1	Operation	
0000	aD = p	p = x*y
0001	aD = aS + p	p = x*y
0010		p = x*y
0011	aD = aS – p	p = x*y
0100	aD = p	
0101	aD = aS + p	
0110	NOP	
0111	aD = aS – p	
1000	aD = aS | y	
1001	aD = aS ^ y	
1010	aS & y	
1011	aS – y	
1100	aD = y	
1101	aD = aS + y	
1110	aD = aS & y	
1111	aD = aS – y	

   The value of S can be 0 to select a0 or 1 to select a1. The value of $\overline{aT}$ can be 0 to select a1 or 1 to select a0. Since aD and aT must be different accumulators, aD will be the opposite of aT. Flags are modified based on the value computed by the ALU. Note: for all diadic operations involving the y register, y is sign-extended to 36 bits before performing the operation (this includes logical operations).

2. Save either the aT or aTl register into an internal temporary location (temp). $\overline{aT}$ is defined as the opposite of D for this instruction. Therefore, if the F1 field selects writing to aD, aD will be the opposite of aT since the $\overline{aT}$ field must read/write aT, and vice versa. Note that if aS in the F1 operation is the same as aT, the value used in the F1 operation will be the old value, due to pipelining. The X field selects aT or aTl:

   X = 0 → aTl   X = 1 → aT

3. Access the internal RAM location pointed to by rN, and write this value to the aT (or aTl) register. rN is specified by the two most significant bits of the Z field:

   00 - r0   01 - r1   10 - r2   11 - r3

4. Postmodify the value of rN by the first action described by the two least significant bits of the Z field (described below).
5. Write the value saved in the temporary register (temp) to the memory location now pointed to by rN.
6. Postmodify the value of rN by the second action described by the two least significant bits of the Z field. The available options for the postmodification are specified as follows:

Symbol	2 LSBs of Z	First Action	Second Action
*rNzp	00	no action	postincrement
*rNpz	01	postincrement	no action
*rNm2	10	postdecrement	postincrement by 2
*rNjk	11	postincrement by (j)	postincrement by (k)

Code 11, in this case, means add the current value of the j (or) k register to rN (after accessing *rN).

Bit	15				11	10	9	8 – 5	4	3 – 0
Field	0	0	1	0	1	$\overline{\text{a}}$T	S	F1	X	Z

Words:	1
Cycles:	2
Group:	Multiply/ALU
Addressing:	Register Indirect, Register
Flags affected:	All
Interruptible:	Yes
Cacheable:	Yes
Format:	2a

F1 Z : y x = *pt++[i]

(multiply/ALU operation with compound data move and parallel load of x register)

```
perform operation F1; then
temp ← (y); then
(y) ← (*rN); then
modify rN (first action); then
(*rN) ← temp; then
modify rN (second action); then
(x) ← (*pt); then
(pt)= (pt)+ [1 or i]
```

This instruction performs the following operations (effectively in sequence):

1. The operation F1 is performed. The possible operations for F1 are:

F1	Operation	
0000	aD = p	p = x*y
0001	aD = aS + p	p = x*y
0010		p = x*y
0011	aD = aS – p	p = x*y
0100	aD = p	
0101	aD = aS + p	
0110	NOP	
0111	aD = aS – p	
1000	aD = aS | y	
1001	aD = aS ^ y	
1010	aS & y	
1011	aS – y	
1100	aD = y	
1101	aD = aS + y	
1110	aD = aS & y	
1111	aD = aS – y	

   The value of S can be 0 to select a0 or 1 to select a1. The value of D can be 0 to select a0 or 1 to select a1. Flags are modified based on the value computed by the DAU. Note: for all diadic operations involving the y register, y is sign-extended to 36 bits before performing the operation (this includes logical operations).

2. Save the y register into an internal temporary location (temp).

3. Access the internal RAM location pointed to by rN, and write this value into the y register. rN is specified by the two most significant bits of the Z field:

   00 - r0   01 - r1   10 - r2   11 - r3

4. Postmodify the value of rN by the first action described by the two least significant bits of the Z field (described below).
5. Write the value saved in the temporary register (temp) to the memory location now pointed to by rN.
6. Postmodify the value of rN by the second action described by the two least significant bits of the Z field. The available options for the postmodification are specified as follows:

Symbol	**2 LSBs of Z**	**First Action**	**Second Action**
*rNzp	00	no action	postincrement
*rNpz	01	postincrement	no action
*rNm2	10	postdecrement	postincrement by 2
*rNjk	11	postincrement by (j)	postincrement by (k)

Code 11, in this case, means add the current value of the j (or) k register to rN (after accessing *rN).

7. Access the ROM location pointed to by pt, and write this value into the x register. Either internal or external ROM may be accessed, depending on the state of the EXM pin (and the address, for the DSP16A).
8. Postmodify the value of the pt register by either 1 or i, selected by the X field:

X = 0 → *pt++ X = 1 → *pt++i

Bit	15				11	10	9	8 – 5	4	3 – 0
Field	1	1	1	0	1	D	S	F1	X	Z

Words:	1
Cycles:	2
Group:	Multiply/ALU
Addressing:	Register Indirect, Register
Flags affected:	All
Interruptible:	Yes
Cacheable:	Yes
Format:	2

Chapter 4
DSP16/DSP16A Device Programming

CHAPTER 4. DSP16/DSP16A DEVICE PROGRAMMING

CONTENTS

4. DSP16/DSP16A DEVICE PROGRAMMING

This chapter discusses various aspects of programming the DSP16 and DSP16A devices. Many of the topics are illustrated in the complete sample programs presented in Appendix B. Techniques for programming the serial and parallel I/O sections of the device may be found in Chapters 5 and 6, respectively.

Chapter 3 described the instruction set specific to the DSP16/DSP16A device. Programming examples in this manual follow the assembler syntax of the *WE* DSP16/DSP16A Support Software Library for DSP16/DSP16A source files. An overview of the DSP16/DSP16A assembly language is provided in the following section.

4.1 DSP16/DSP16A ASSEMBLY-LANGUAGE NOTATION

A DSP16/DSP16A source file exists as a text file and contains DSP16/DSP16A instructions, directives to allow the assembler to interpret the instructions and data, and comments to clarify the use of the program. The syntax of the assembler directives is described in this chapter; also described are conventions and nomenclature used throughout the remainder of this manual. Appropriate formats for DSP16/DSP16A source files are also discussed.

4.1.1 Integer Notation

Decimal, hexadecimal, or octal expressions may be freely mixed when specifying numerical data in a source file. The syntax is identical to C-language programming.

- **Decimal**. Any string of normal digits (0—9) is interpreted as a decimal number, provided it does not have a leading zero.
- **Hexadecimal**. A numerical string beginning with 0x is interpreted as a hexadecimal number and may contain the digits 0—9, a—f, or A—F. For example, 0x0 is the same as 0. 0x010 is the same as 0x10, which is the decimal number 16. And 0xFF or 0xff is the decimal number 127.
- **Octal**. A numerical string beginning with the digit zero is interpreted as an octal number and may contain the digits 0—7. For example, 07 is the decimal number 7 and 010 is the decimal number 8.
- **Fixed-Point**. Numbers with a decimal point are interpreted as binary fixed-point numbers by the DSP16/DSP16A assembler. The number of binary digits to the right of the decimal point is 14 by default, but may be changed (see the *WE® DSP16 and DSP16A Support Software Library User Manual*).

4.1.2 Comments

Comments may be placed in the source file to enhance readability and to provide information for other users. A comment may be placed on a line by itself or may appear at the end of a line containing an instruction. The following lines are examples of valid comments:

```
/* This is a valid comment */
instruction
instruction          /* this is a valid comment */
```

4.1.3 Labels

Labels in a source file serve two purposes: to give a descriptive name to a particular location and to provide a destination for a branch instruction. Labels may consist of upper- and/or lower-case alphanumeric characters and the underscore, although the first character may not be numeric. A label must be terminated with a colon. Labels may be as long as necessary to be descriptive; however, only the first eight characters are significant. The following lines show examples of valid labels:

```
start_1:  instruction     /* "start_1" is a valid label */
          instruction
    end:  instruction     /* "end" is a valid label */
```

4.1.4 Data Stored in ROM

Data may be stored in a ROM location by using the int directive. The following lines of source code are examples of how to store data in ROM:

```
   table:  int     0xFF          /* Initialize one ROM location    */
           2*int   0x10 0xA2     /* Initialize four ROM locations  */
           3*int   23            /* Initialize three ROM locations */
   fixed:  int     1.23 -1.634   /* Initialize two ROM locations   */
 tab_end:  int     3.721!10      /* Initialize one ROM location    */
```

As shown above, multiple ROM locations may be specified with a single statement. In the second example, two ROM locations are replicated to initialize four ROM locations; however, in the third example, all three locations are initialized to the same value.

Following the label *fixed*, two ROM locations are initialized in a fixed-point notation. By default, the DSP16/DSP16A assembler assumes that fixed-point numbers are to be assembled with 14 bits of precision and 2 bits of magnitude. An environment variable, precision, may be changed to allow other values of precision.

The last example demonstrates another method to specify the precision of a fixed-point format. The suffix !N (where N is the desired precision) can be used to force different precision encodings "on the fly." In this case, 3.721 is encoded with 6 magnitude bits and 10 precision bits.

4.1.5 RAM Variables

RAM variables can be allocated similarly to data stored in ROM by surrounding the int directives with the .ram and .endram directives. Note that RAM locations are allocated without being initialized. The following sequence allocates six RAM variables:

```
.ram
data1:   int       /* allocate 1 RAM variable  */
data2:   2*int     /* allocate 2 RAM variables */
data4:   3*int     /* allocate 3 RAM variables */
.endram
```

4.1.6 DSP16/DSP16A Source-File Format

A DSP16/DSP16A source file is prepared as a text file by using a text editor with the *UNIX* Operating System or *MS-DOS* Operating System. (See Appendix B for complete DSP16/DSP16A program listings.) When creating a source file, the following conventions should be observed:

- The source file name must end with ".s".
- Directives beginning with "." (such as .ram and .endram) must begin in the first column.
- White space is used to separate the fields of instructions. Either a space or a tab character constitutes white space. Using tabs to separate and align the fields improves the readability of source files.
- Labels normally begin in the first column to enhance readability, but may be indented if desired.
- It is customary, but not required, to place the title and a brief description of the program at the top of the file for reference.

4.2 PROGRAMMING TECHNIQUES

The following sections describe problems commonly encountered when programming the DSP16/DSP16A device and their possible solutions. In general, many of the problems encountered when programming other digital signal processors (such as latency and pipeline effects) have been eliminated by the design of the DSP16 and DSP16A devices.

4.2.1 Instruction Set Ambiguities

Several instructions, which normally would be written identically, can be interpreted as various types of instructions. This interpretation of the instructions determines the number of ROM locations used to store the instruction, the number of instruction cycles used to execute the instruction, and whether or not the instruction affects the flags. Hence, the interpretation can be critical. For example, the instruction

```
a0 = y
```

could be a multiply/ALU, special function, or data move instruction. When the instruction is interpreted as a multiply/ALU or special function instruction, the instruction requires one ROM location and executes in one instruction cycle. When the instruction is interpreted as a data move instruction, the instruction requires one ROM location and executes in two instruction cycles. The interpretation of the instruction is critical if conditional testing based on the results of the instruction execution is performed. The DSP16/DSP16A flags are affected by the multiply/ALU and special function instructions, but not by the data move instructions.

The *WE* DSP16/DSP16A Support Software Library provides optional mnemonics that may be used with an instruction to specify its type. Table 4-1 shows the mnemonics that can be used to specify the type of instruction. For example, the instruction

```
au  a0 = y
```

is interpreted as a multiply/ALU instruction.

Table 4-1. Optional Mnemonics

Use	To Specify
au	Multiply/ALU instruction
if true	Special function instruction
set	Short immediate instruction
move	Data move instruction

If an instruction may be encoded several ways, the assembler chooses the encoding based on the following priority:

1. Special function
2. Multiply/ALU
3. Short immediate
4. Data move

4.2.2 Polling for I/O

When not using interrupt driven I/O, polling for input and output conditions is the simplest means of handling I/O timing. The following segment of code continuously polls the pioc register to determine if the condition IBF is true, meaning that there is data in the serial input register waiting to be processed. When data is loaded into the serial input buffer from an external device, program execution continues below the wait loop.

```
        y = 0x010           /* place mask into y register  */
wait:   a0 = pioc           /* check pioc register for IBF */
        a0 & y              /* - look only at bit 4        */
        if eq goto wait     /* - if no input, wait.        */
             .
             .
        *r0 = sdx           /* move data into RAM */
```

This same code fragment can be used to poll any I/O condition by changing the value in register y, which is used to mask the unwanted bits of the pioc register. For example, use 0x04 to check only the condition PIDS, which indicates that a parallel input was performed.

4.2.3 Modulo Addressing

Modulo addressing is provided to allow efficient implementation of cyclical memory accesses. To use modulo addressing, the first RAM address of the modulo must be loaded into register rb and the last RAM address into re. The register being used as the memory pointer must be postincremented by +1. Each time the pointer is used, its value is compared with the contents of register re (before the postincrement is performed). If the two values are equal, the value of register rb is loaded into the register being used to address the RAM and the cycle repeats.

It is important to note that whenever register re contains a value not equal to zero, modulo addressing is active. On reset, the value of re is zero. Whenever modulo addressing is not used, this register should contain zero and should not be used to store any number other than the address of the end of a modulo.

4.2.4 Random Number Generation

The DAU includes a 10-state pseudorandom binary sequence (PRBS) generator, which is used to toggle a bit in the DAU. The status of this bit may be determined by testing for the "heads" or "tails" condition. The following segment of code generates a 16-bit random number in the high half of accumulator a0 by randomly setting each of the 16 bits:

```
do 16 {
   if heads a0h = a0h + 1      /* if heads, set bit to 1 */
   a0 = a0 << 1                /* shift left 1 position  */
}
```

The pseudorandom sequence is incremented each time it is tested and may be reset by writing any value to the pi register (writing to the pi register does not affect its contents except when in an interrupt service routine). (See Section 4.2.5.)

4.2.5 Programming Tips

The following section describes several practical programming tips that may not be obvious to a new user of the DSP16/DSP16A.

1) When loading count values into c0 and c1, the count value is 1 − *count*, where *count* is the desired number of times the loop is to be executed. An easy way to assemble the loop counter load is to let the assembler compute the 1 − *count* value. For example, if a loop is to be repeated 10 times, the following code could be used:

```
         c0 = 1 - 10
   loop: ...
         if c0lt goto loop
```

   The assembler correctly computes 1 − count, and the code is easier to read.

2) If extra 16-bit registers are needed, there are several possible ways to "create" them.

   a) If not using interrupts or development system breakpoints, an **icall** instruction may be placed at location 0. This causes a branch to location 2 (where program execution begins) and makes the DSP16/DSP16A "think" that it is in an interrupt service routine (ISR). While in an ISR, the DSP16/DSP16A no longer updates the pi register each time the pc register changes, and the pi register may be written to (writes to pi do not affect its contents when not in an ISR, but writing the pi register resets the pseudorandom sequence generator). When in an ISR, the pi register is not used by the DSP16/DSP16A and is free for use as a general-purpose 16-bit register.

   b) While not in a subroutine, the pr register is available as a general-purpose 16-bit register.

   c) While not doing ROM table lookups, the pt register is available as a general-purpose 16-bit register. It can easily be incremented or modified using:

```
{y = Y, Y = aT, Z : y}  x = *pt++    /* load of y necessary when */
                                     /* loading x from ROM       */
```

   or

```
i = N                                /* or -N */
{y = Y, Y = aT, Z : y}  x = *pt++i
```

   Note: The XAAU adder is only 12 bits wide, therefore, modifying as above is modulo 4K, i.e.,

```
pt = 4095
{y = Y, Y = aT, Z : y}  x = *pt++
                                   /* pt is now 0,      */
                                   /* not 4096 (2**12) */
```

   However,

```
a0 = pt
a0h = a0h + 1
pt = a0
```

   is no problem, except above 32767 unless saturation logic is disabled on a0 (since a value above 32767 appears to be an overflowed 2's complement value).

3) While not using modulo addressing (re = 0), the rb register is available as a general-purpose register. The re register is not available since a non-zero value enables modulo addressing. Note that all YAAU register are 9 bits wide in the DSP16 and 16 bits wide in the DSP16A.

4) If a write of 0 to a RAM location is required, and modulo addressing is not being used, the re register can be used (re is zero by definition).

```
*rN [++, --, ++j] = re
```

   clears the RAM location pointed to by rN with no setup required.

5) If adding and subtracting accumulators without using y is desired, the following instructions could be used to perform an add (assuming that only the high half of an accumulator is being used or the high half is a whole number and the low half is a fraction):

```
a0 = a0 >> 4
a0h = a0h + 1
a0 = a0 << 4        /* adds 16 (2**4) to a0        */
                    /* (similarly, << 8 adds 256)  */

a0 = -a0
a0h = a0h + 1
a0 = -a0            /* subtracts 1 from a0 */
```

Shifting left and adding can be used to add fractions.

6) The following two-cycle data move instructions can be coded as a single-cycle multiply/ALU instruction (when executing in the cache) by doing a dummy load to x.

```
do 40 {
      y = aN        /* 2-cycle data move */
      }
```

This takes 81 machine cycles, while:

```
do 40 {
      y = aN  x = *pt++     /* single-cycle when in cache */
      }
```

takes only 43 machine cycles (2 when it is loaded the first time and 2 the last time it is executed). In both cases, the **do** instruction requires 1 cycle. Note that this is a trivial example to make the cycle counts more obvious. This "trick" is most useful when the kernel of an operation is in the cache and a result needs to be multiplied by a coefficient or operated on by the ALU.

7) The above assumes that pt has already been set and that postincrementing pt does not affect anything. If this is not true (postincrementing pt is not desired), the following can be done:

```
i = 0
do 40 {
      y = aN  x = *pt++i        /* postincrement by 0 */
      }
```

This does not alter the value of pt.

4.2.6 Concurrent Interrupts

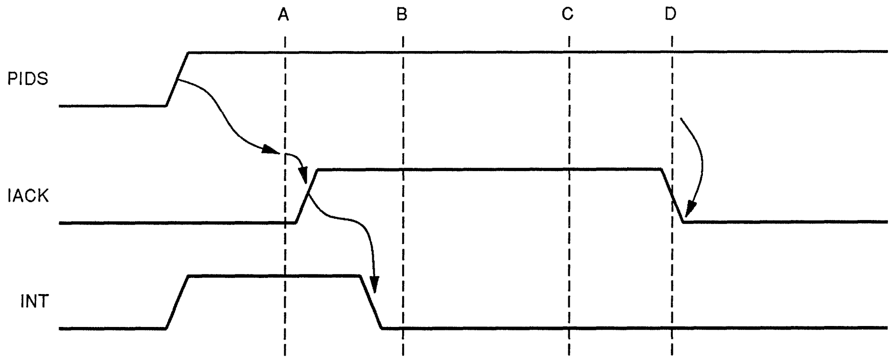
Consideration must be given to situations in which multiple interrupting conditions occur. The DSP16/DSP16A device does not allow nesting of interrupts; however, there are other ways to guarantee that all interrupts can be recognized and serviced.

Case 1

If an internal and external interrupt request occur at nearly the same time and before the execution of the branch-to-one (start of interrupt service routine), the status field in the pioc register can be examined. In this case, the status will indicate that both interrupts are pending. They can be serviced accordingly.

RULE: An interrupt occurring after an internal interrupt occurs and before IACK is asserted (in response to the internal interrupt) causes the INT bit in the pioc register (bit 0) to be set, providing that INT meets its assertion time requirements.

The INT signal is negated on the rising edge of IACK.



A. Branch-to-one instruction executed. Beginning of interrupt service responding to negation of PIDS.

B. pioc register has PIDS and INT status bits set.

C. ireturn instruction executed. End of interrupt service routine.

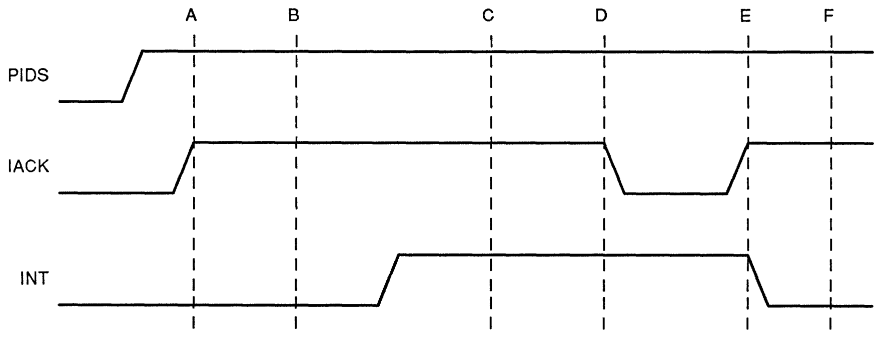
D. Next interruptible instruction.

Figure 4-1. Case 1 – Internal Interrupt (PIDS) and INT Occur Before Assertion of IACK

Case 2

If INT is asserted (high) when IACK is already asserted (i.e., when the DSP16 device is servicing another interrupt), then INT must remain asserted until the next rising edge of IACK. This is because the internal interrupt request is cleared on the falling edge of IACK. This guarantees that the interrupt request (assertion of INT) will be serviced at the next interruptible instruction after the currently executing interrupt service routine has finished.

RULE: To guarantee recognition of INT when it is asserted during an interrupt service routine (IACK high), INT should not be negated until the next rising edge of IACK, providing that INT assertion time is met.



A. Branch-to-one instruction executed in response to internal interrupt (PIDS).

B. pioc register has PIDS status bit set.

C. ireturn instruction executed.

D. Next interruptible instruction.

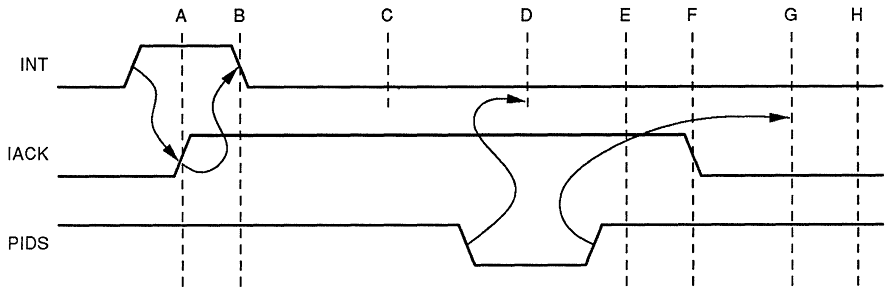
E. Branch-to-one instruction executed in response to INT.

F. pioc register has INT status bit set.

Figure 4-2. Case 2 – INT Asserted During Service of Internal Interrupt After pioc Status is Checked

Case 3

Internal interrupt requests remain pending until the respective pdx or sdx registers are serviced. Hence, if an external interrupt is being serviced and another internal interrupt request is generated, the internal interrupt request remains pending and causes a second interrupt to be taken at the next interruptible instruction. In this way, the internal interrupt is not missed if it occurs during the servicing of another external interrupt.



A. Branch-to-one instruction.

B. pioc register has INT status bit set.

C. Read of pioc register status.

D. pioc register has PIDS status bit set.

E. ireturn instruction executed.

F. Next interruptible instruction.

G. Branch-to-one instruction. Begin to service internal interrupt.

H. Service internal interrupt.

Figure 4-3. Case 3 – Internal Interrupt Asserted While Servicing an External Interrupt

Case 4

If it is possible for two or more interrupt requests to be pending, the easiest method for servicing these interrupts is to service the external interrupt first and then the internal interrupt requests individually (by taking a new interrupt for each internal request) until no more interrupts are pending. The drawback of this procedure is that if external interrupts are frequent, there may be a large latency when servicing internal interrupts.

4.2.7 Interrupt Latency

Two classes of DSP16/DSP16A instructions are not interruptible. The first class contains all branch instructions. The second class contains instructions that are executing in the cache (i.e., any instruction when executing from the on-chip instruction cache cannot be interrupted).

Interrupt latency is bounded by the longest in-cache operation. In situations where interrupt latency is critical, in-cache operations should be split into smaller cache operations whose execution time is less than any critical latency requirements. In this situation, an interruptible instruction must be placed between successive cache instructions.

For example:

```
do 93 {
        instr
        instr
          .
          .
          .
        instr
       }
nop                         /* interruptible instruction */
redo 50
nop                         /* interruptible instruction */
redo 50
```

Chapter 5
Serial I/O

CHAPTER 5. SERIAL I/O

CONTENTS

5. SERIAL I/O

The serial I/O port on the DSP16/DSP16A device provides a serial interface to many codecs and signal processors with few if any external chips. The high-speed, double-buffered port supports back-to-back transmissions. The output buffer empty (OBE) and input buffer full (IBF) flags facilitate the reading and/or writing of the serial I/O port by program or interrupt driven I/O. There are four selectable active clock speeds. A bit-reversal mode provides compatibility with either most significant bit (MSB) first or least significant bit (LSB) first serial I/O formats. A multiprocessor I/O configuration requiring no external chips is provided. Three registers, serial I/O control (sioc), time-division multiplexed slot (tdms), and serial receive/transmit address (srta), allow the modes of operation to be controlled.

Figure 5-1 shows a simplified block-level representation of the serial I/O data path. The double-buffered inputs (isr and ibuf) and outputs (obuf and osr) connect to the internal data bus. The serial I/O uses a register-based implementation. The input and output buffer registers (ibuf and obuf, respectively) are used to input and output the data through the port. Both registers are referenced in the instruction set by the name sdx. Unlike other registers in the DSP16/DSP16A device, the writing of sdx and the reading of sdx are performed on two distinct registers. The ICK, OCK, ILD, and OLD interface is represented by the clock generator block. The signals connected to this block are bidirectional and may be programmed via the sioc register. The ifsr and ofsr provide flag signals for the input and output (IBF and OSE), respectively. The multiprocessor I/O is not represented in Figure 5-1. The signals shown on the lower portion of Figure 5-1 are described in Section 5.3.

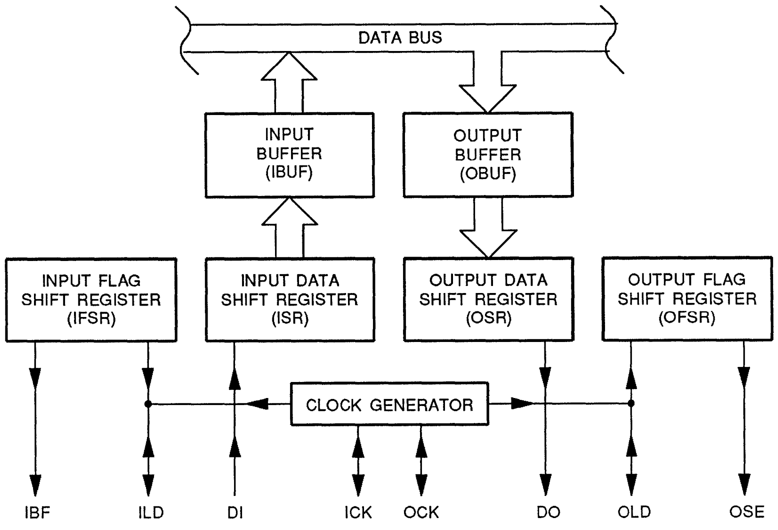

DATA BUS
INPUT BUFFER (IBUF)
OUTPUT BUFFER (OBUF)
INPUT FLAG SHIFT REGISTER (IFSR)
INPUT DATA SHIFT REGISTER (ISR)
OUTPUT DATA SHIFT REGISTER (OSR)
OUTPUT FLAG SHIFT REGISTER (OFSR)
CLOCK GENERATOR
IBF
ILD
DI
ICK
OCK
DO
OLD
OSE

Figure 5-1. Serial I/O Internal Data Path

5.1 SIO OPERATION

The following subsections describe the operation of the SIO input and output sections and the active clock generator.

5.1.1 Input Section

A typically free-running clock (ICK) synchronizes all events occurring within the input section of the SIO. A high-to-low transition of the input load (ILD) signal followed by a rising edge of ICK initiates the start of an input transaction. The first serial data bit is read from DI on the next rising edge of ICK. Eight or sixteen bits later, when the input shift (isr) register fills, this data is transferred to the input buffer (ibuf) register and the input buffer full (IBF) signal is asserted, indicating that the buffer is full. The DSP16/DSP16A device may read the data at this time. The read command is of the type a0 = sdx, a1 = sdx, or *Y* = sdx. The IBF is negated when the input buffer is read. Another serial input may begin before the input buffer read takes place since the port is double-buffered. If the new transfer is completed before the previous input is read, then the new data is transferred to the input buffer, overwriting the old data. The status of IBF may be read from the pioc register, IBF status field (bit 4), or the IBF field (bit 15) – this is the sign bit that can be tested without masking
(i.e., a0 = pioc \ a0 = a0 \ if pl goto loop). The IBF may also be used as an interrupting condition, if the appropriate enable bit in the pioc register is set (pioc register, bit 9).

5.1.2 Output Section

When the DSP16/DSP16A device is reset (power-up or RSTB), the internal status flag output buffer empty (OBE) is set, indicating that the buffer is empty. When data is written to the output buffer by an instruction of the form sdx = a0, sdx = a1, sdx = *Y*, or sdx = *value*, OBE is cleared and the serial output section is ready for a serial transmission. The status of the OBE flag may be read from the pioc register (OBE status field, bit 3). The OBE may be used as an interrupting condition if the appropriate enable bit in the pioc register is set (pioc register, bit 8). A typically free-running clock (OCK) synchronizes all events taking place within the output section. A high-to-low transition of the output load (OLD) signal, followed by a rising edge of OCK, initiates the start of an output transaction. This procedure causes the contents of the output buffer register to be transferred to the output shift (osr) register, the internal flag OBE to be set (indicating the need for more data), and a high-to-low transition of OSE (indicating that the shift register is full). The first serial data bit is placed on the data output (DO) at this time. Eight or sixteen bits later, when the serial output has been completed, the output shift register empty (OSE) signal will be asserted, indicating that the last bit of the serial transmission has been sent. (OSE can be used by external hardware to latch a shift register.) If the output buffer has been reloaded, another transfer begins immediately; otherwise, zeros are sent on the serial output until the buffer is reloaded prior to a high-to-low transition of OLD beginning another transmission. Double-buffering allows the output buffer to be reloaded while data is being shifted out of the output shift register.

A serial address (SADD) transmits simultaneously with DO. This address is the transmit address field of the srta register (see Table 5-4). The SADD output is active low. The SADD may also be used as a second serial output (only) port. The SADD signal is valid whether or not the device is in the multiprocessor mode (See Section 5.6). If SADD is to be used in this manner, the LD field of the sioc register should be set high to synchronize SADD with DO.

5.1.3 Active Clock Generator

The active clock signals for the SIO section are derived from CKI, with a maximum frequency of CKI ÷ 4. A simplified representation of the SIO active clock and load generator is shown in Figure 5-2. In the figure, the switches represent the user-programmable features. A closed switch corresponds to the associated bit in the sioc or tdms register having a value of one.

The five signals ICK, OCK, ILD, OLD, and SYNC can be individually programmed to be either inputs or outputs (passive or active). When using active clocks (generated by the DSP16/DSP16A device), the speed of the clocks can be selected from four speeds: CKI divided by 4, 12, 16, or 20. This selection determines the speed of both ICK and OCK. The speed of ILD and OLD can be selected as either the ICK or OCK signals divided by 16. An active SYNC signal is generated from this same source (ICK or OCK divided by 16) and is further divided by 8 or 16. The resulting SYNC signal is either the signal ICK or OCK divided by 128 or 256. The SYNC signal can be configured to generate an 8 kHz sampling signal for codec applications.

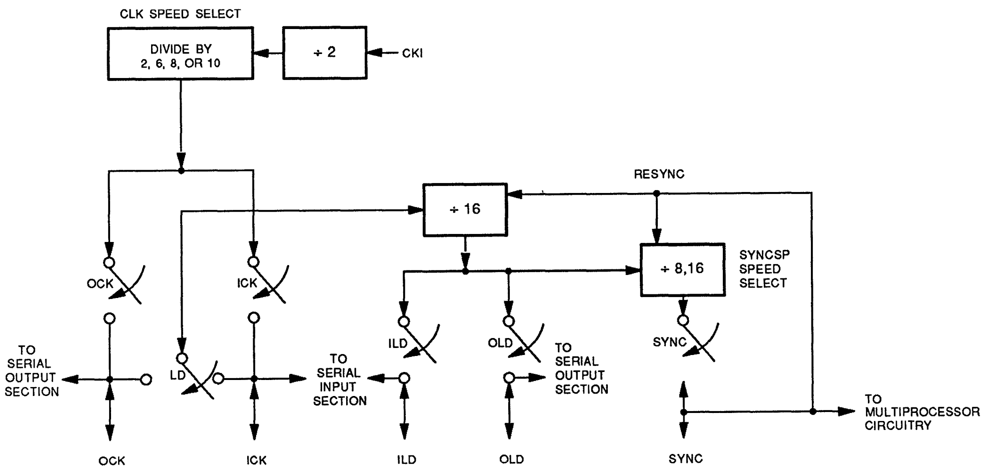


Figure 5-2. SIO Active Clock and Load Generation

5.2 USER-CONTROLLED FEATURES

Programmable modes are controlled by the serial I/O control (sioc) register. Flexibility in programming the functions of the serial I/O port allows the port to interface with a variety of devices with little or no "glue logic." Table 5-1 shows the control bits of the sioc register. During device reset, the sioc register bits are cleared.

Table 5-1. Serial I/O Control (sioc) Register

Bit	9	8 7	6	5	4	3	2	1	0
Field	LD	CLK	MSB	OLD	ILD	OCK	ICK	OLEN	ILEN

Field	Value	Result/Description
LD	0	Active ILD/OLD = ICK÷16, Active SYNC = ICK÷128/256.†
	1	Active ILD/OLD = OCK÷16,‡ Active SYNC = OCK÷128/256.†,‡
CLK	0 0	Active clock = CKI÷4.
	0 1	Active clock = CKI÷12.
	1 0	Active clock = CKI÷16.
	1 1	Active clock = CKI÷20.
MSB	0	LSB first.
	1	MSB first.
OLD	0	OLD is an input (passive mode).
	1	OLD is an output (active mode).
ILD	0	ILD is an input (passive mode).
	1	ILD is an output (active mode).
OCK	0	OCK is an input (passive mode).
	1	OCK is an output (active mode).
ICK	0	ICK is an input (passive mode).
	1	ICK is an output (active mode).
OLEN	0	16-bit output.
	1	8-bit output.
ILEN	0	16-bit input.
	1	8-bit input.

† Either 128 or 256 – see tdms register SYNC field.
‡ Select this mode when using SADD (not necessary if ICK = OCK).

The following section describes each programmable mode in detail.

- **LD** – The LD field (sioc bit 9) allows the active (internally generated) ILD and OLD signals to be derived from either ICK (LD = 0) or OCK (LD = 1). Active ILD and OLD always are derived from the same source.
- **CLK** – The CLK field (sioc bits 8,7) allows one of four active I/O speeds to be selected: a division of the input clock CKI by either 4, 12, 16, or 20. As an example, with a CKI of 8.192 MHz, 24.576 MHz, 32.768 MHz, or 40.960 MHz, using the appropriate divisor of 4, 12, 16, or 20, respectively, results in an active I/O rate of 2.048 MHz. Refer to Table 5-1 for the CLK field encoding.
- **MSB** – The MSB field (sioc bit 6) determines the bit order of the serial transmissions: most significant bit (MSB) first (MSB = 1) or least significant bit (LSB) first (MSB = 0). This mode switch allows compatibility with devices that perform either MSB first or LSB first serial transfers. This mode is also useful when performing µ-law or A-law conversions. A minimal amount of software is required to perform these conversions. Since this field allows the bit order to be switched when an sdx read or write occurs, the MSB field can be switched immediately before and/or after an sdx read or write. If this technique is used in other than an interrupt service routine, care should be taken to insure that the proper mode is in effect in the event of an interrupt.
- **OLD** – The OLD field (sioc bit 5) allows OLD to be either an input (OLD = 0) or an output (OLD = 1).
- **ILD** – The ILD field (sioc bit 4) allows ILD to be either an input (ILD = 0) or an output (ILD = 1).
- **OCK** – The OCK field (sioc bit 3) allows OCK to be either an input (OCK = 0) or an output (OCK = 1).
- **ICK** – The ICK field (sioc bit 2) allows ICK to be either an input (ICK = 0) or an output (ICK = 1).
- **OLEN** – The OLEN field (sioc bit 1) controls the length of the serial output: either 16-bit (OLEN = 0) or 8-bit (OLEN = 1). When the data is sent in the 8-bit mode with the LSB first (MSB = 0), the eight data bits should be placed in the least significant half of obuf; i.e., 0x00DD (D = data). When the data is sent in the 8-bit mode with the MSB first (MSB = 1), the eight data bits should be "packed" in the most significant half of obuf; i.e., 0xDD00 (D = data).
- **ILEN** – The ILEN field (sioc bit 0) controls the length of the serial input: either 16-bit (ILEN = 0) or 8-bit (ILEN = 1). When the data is sent in the 8-bit mode with the LSB first (MSB = 0), the eight data bits are placed in the most significant half of ibuf; i.e., 0xDD00 (D = data). When the data was sent in the 8-bit mode with the MSB first (MSB = 1), the eight data bits are placed in the least significant half of ibuf; i.e., 0x00DD (D = data).

5.3 SERIAL I/O PIN DESCRIPTIONS

The physical serial I/O port consists of eleven signals: four are used for serial input, four are used for serial output, and three are used in multiprocessor and/or TDM applications. Table 5-2 lists each signal with its type, pin number, and description.

Table 5-2. Serial I/O Pins

Symbol	Type*	Pin	Name/Description
DI	I	56	**Data Input**. Serial PCM data latched on rising edge of ICK, either LSB or MSB first, according to the sioc register MSB field.
ICK	I/O†	58	**Input Clock**. Clock for serial PCM input data. In active mode, ICK is an output; in passive mode, ICK is an input, according to the sioc register ICK field.
ILD	I/O†	57	**Input Load**. Falling edge of ILD indicates the beginning of a serial input word. In active mode, ILD is an output; in passive mode, ILD is an input, according to the sioc register ILD field.
IBF	O†	53	**Input Buffer Full**. IBF is asserted when the input buffer is filled and negated by a read of the buffer. IBF is also negated by asserting RSTB.
DO	O†	61	**Data Output**. Serial PCM data output from the output shift register (osr), either LSB or MSB first – according to the sioc register MSB field. DO changes on the rising edges of OCK. DO is 3-stated when DOEN is high.
DOEN	I/O†	64	**Data Output Enable (Active-Low).** An input when not in the multiprocessor mode. DO and SADD are enabled only if DOEN is low. DOEN is an output when in the multiprocessor mode (tdms register MODE field set). In the multiprocessor mode, DOEN indicates a valid time slot for a serial output.
OCK	I/O†	59	**Output Clock**. Clock for serial PCM output data. In active mode, OCK is an output; in passive mode, OCK is an input, according to the sioc register OCK field.
OLD	I/O†	60	**Output Load**. Clock for loading the parallel-to-serial converter from the output buffer (obuf). A falling edge of OLD indicates the beginning of a serial output word. In active mode, OLD is an output; in passive, OLD is an input, according to the sioc register OLD field.
OSE	O†	52	**Output Shift Register Empty**. Indicates the end of a serial transmission. OSE is set either by the emptying of the output shift register or by asserting RSTB. OSE is reset by the DSP16 writing a word (two clock cycles after the falling edge of OLD) to the output shift register. If no new word is written by the DSP16, OSE remains high regardless of activity on OLD.

* I = Input; O = Output.
† 3-stated.

Table 5-2. Serial I/O Pins (continued)

Symbol	Type*	Pin	Name/Description
SADD	I/O†	63	**Serial Address (Active-Low).** An 8-bit serial bit stream typically used for addressing during multiprocessor communication between multiple DSP16 devices. In multiprocessor mode, SADD is an output when the tdms time slot dictates a serial transmission; otherwise, it is an input. SADD is always an output when not in multiprocessor mode and can be used as a second serial output. SADD is 3-stated when DOEN is high.
SYNC	I/O†	62	**Multiprocessor Synchronization.** Typically used in the multiprocessor mode, a falling edge of SYNC indicates the first word of a TDM I/O stream and causes the resynchronization of the active ILD and OLD generators. SYNC is an output when the tdms register SYNC field is set; otherwise, it is an input. SYNC must be tied low if it is not used as an output. When used as an output, SYNC = ILD/OLD ÷ 8 or 16, depending on the setting of the SYNCSP field of the tdms register. This procedure can be used to generate a slow clock for SIO operation.

* I = Input; O = Output.
† 3-stated.

5.4 CODEC INTERFACE

Figure 5-3 is the schematic showing the connections required to interface the DSP16/DSP16A device to an AT&T T7500 μ-law/A-law Codec. Figure 5-4 shows the connections necessary to interface the DSP16/DSP16A device to an AT&T T7520 or T7522 High-Precision Codec. In both examples, the SYNC signal is actively driving ILD, OLD, and the codec with an 8 kHz signal.

5.5 SERIAL I/O PROGRAMMING EXAMPLE

The program segment shown in this section demonstrates the use of the serial I/O port's interrupt facility. The advantage of using the interrupt on input buffer full (IBF) is that the input data is read in immediately, making careful placement of the sdx read commands within the program unnecessary. This program allows 128 inputs to be read into a buffer while another buffer already loaded with data is used by the program. When the first buffer fills, the two buffers are switched and the process repeats.

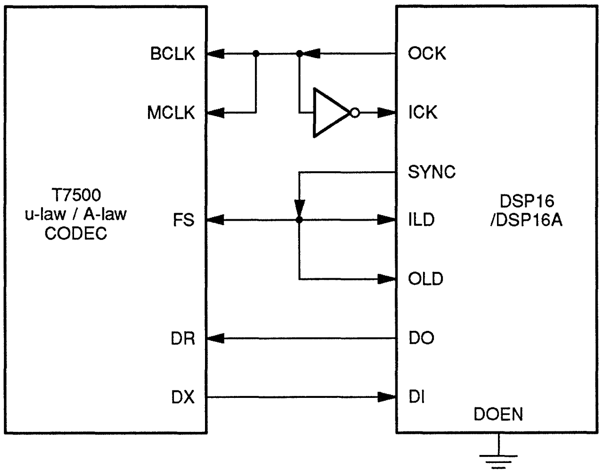


Figure 5-3. ***WE*® DSP16/DSP16A to AT&T T7500 Codec Interface**

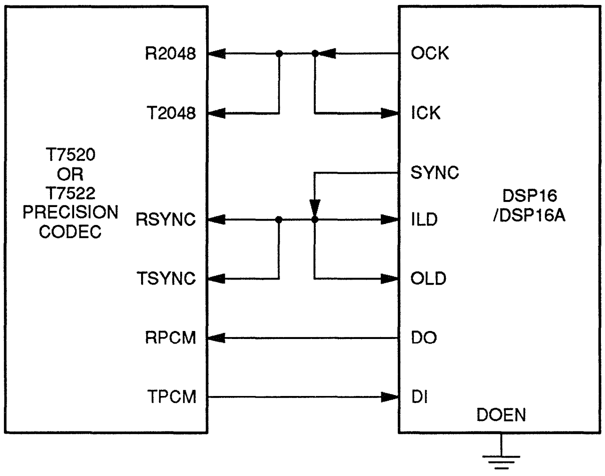


Figure 5-4. ***WE*® DSP16/DSP16A to AT&T T7520 or T7522 Codec Interface**

5.5.1 Program Segment

```
     /*    Ping pong I/O routine     */

           goto start
intrpt:    *r0++ = sdx
           ireturn
start:     auc = 0x0
           pioc = 0x200    /* interrupt on IBF      */
           sioc = 0x0      /* passive I/O           */
           y = 0x0
           r1 = 0xfe
           *r1 = y           /* initialize flag           */
           r2 = 0xfd         /* temp storage              */
           r0 = 0xff         /* interrupt pointer         */
           r1 = 0xff         /* program I/O pointer       */
           y = 0x17f         /* address of last sample in */
           *r2 = y           /* 128 point buffer          */
           goto loop
mainprg:

     /*    Main program here; prog. must take less time than I/O! */

           a0 = *r1++         /* read in data from buffer */

loop:      a0 = r0            /* r0 is address of input ptr.  */
           y = *r2
           a0 - y             /* check for 128 samples in buffer */
           if ne goto loop    /* loop if not full                */
           r1 = 0xfe
           a0 = *r1           /* get alternate buf flag          */
           a0 = a0            /* set DAU flags                   */
           if eq goto buf     /* alternate between buffers       */
           y = 0x00
           *r1 = y            /* set flag for buf1               */
           r0 = 0xff          /* interrupt pointer to buf1       */
           r1 = 0x17f         /* program I/O pointer to buf2     */
           y = 0x17f
           *r2 = y            /* address of last sample in buf1  */
           goto mainprg
buf:       y = 0x01
           *r1 = y            /* set flag for buf2               */
           r0 = 0x17f         /* interrupt pointer to buf2       */
           r1 = 0xff          /* program I/O pointer to buf1     */
           y = 0x1ff          /* address of last sample in buf2  */
           *r2 = y
           goto mainprg
```

5.6 MULTIPROCESSOR MODE DESCRIPTION

The multiprocessor mode allows up to eight DSP16/DSP16A devices to be connected together in such a way as to provide data transmission to any of the individual DSP16/DSP16As in the system. Two registers associated with the multiprocessor mode are the time division multiplexed slot (tdms) register (see Table 5-3) and the serial receive and transmit address (srta) register (see Table 5-4). This mode requires no external hardware and uses a TDM interface with eight time slots per frame. A serial address on the SADD line is sent simultaneously with data on DO from any one device in a predetermined time slot, and the data is received only by other device(s) having the address specified. Each device has a user-programmable receive address associated with it.

In the multiprocessor mode, the following pins are connected together to form a four-wire bus, as shown in Figure 5-5. The DI and DO form a single-wire data bus referred to as DATA; ICK and OCK form a clock line referred to as CK; the SADD forms a single-wire address bus referred to as ADD; and SYNC provides a synchronization line referred to as SYN. Typically, CK and SYN are specified statically for one particular DSP16/DSP16A device to always generate, although CK may also be generated by an external clock. The signals are generated by the DSP16/DSP16A device having active SYNC and OCK signals, which occur when the tdms register SYNC field is set and the sioc register OCK field is set. The other devices use the SYNC and OCK signals in the passive mode to synchronize operations. All DSPs have their ILD and OLD signals in active mode. A high-to-low transition of SYNC delineates transmit slot 0. Eight words are exchanged within a SYNC frame, so the tdms register should have the SYNCSP field set low when in the multiprocessor mode. This provides 128 active ICK and/or OCK cycles per SYNC frame (8 words × 16-bits/word). The multiprocessor mode is turned on by setting the tdms mode field to one.

In the multiprocessor mode, each device can send data in a unique time slot designated by the tdms register transmit slot field. The tdms register has fully decoded fielding in order to allow for one DSP16/DSP16A device transmitting in more than one time slot. This procedure is useful for multiprocessor systems with less than eight DSP16/DSP16A devices, when a higher bandwidth is necessary between certain devices in that system. Each device also has an address specified by the srta register transmit address field (Table 5-4), used to transmit the information regarding the destination of the data and an address assignment made to it by the receive address field referring to its own identity. In subsequent examples, the srta register receive address will be referred to as the "device number." **Note:** It is possible to assign more than one receive address or a duplicate receive address to a DSP16/DSP16A device, but the examples given assume a unique receive address.

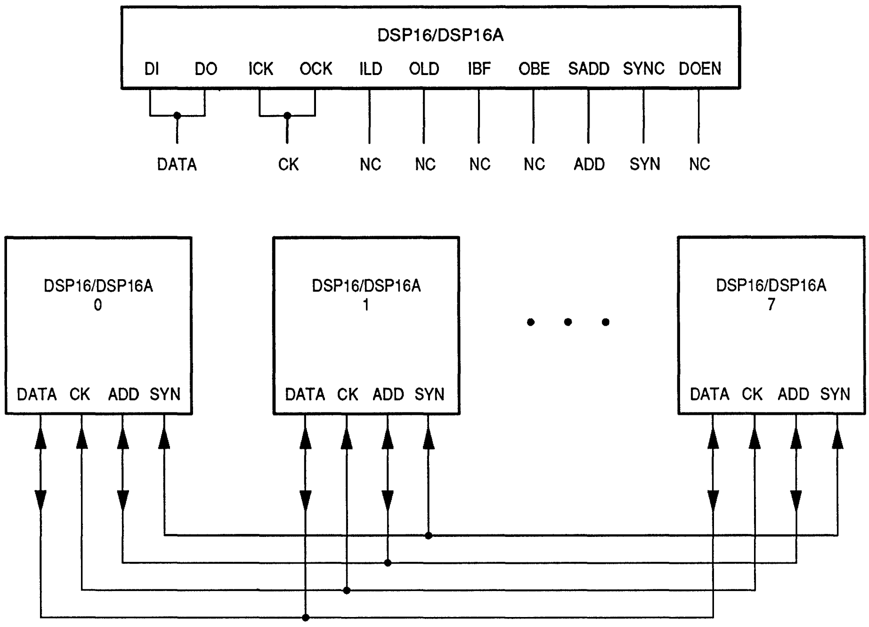

DSP16/DSP16A
DI DO ICK OCK ILD OLD IBF OBE SADD SYNC DOEN
DATA CK NC NC NC NC ADD SYN NC
DSP16/DSP16A 0
DSP16/DSP16A 1
DSP16/DSP16A 7
DATA CK ADD SYN

Figure 5-5. DSP16/DSP16A Multiprocessor Connections

If the serial address coming from the bidirectional SADD line of the transmitting device matches the address of one of the other devices, the input is loaded into that device's input buffer and its IBF flag is set at the end of the transmission. In order to read in the new data, an interrupt could take place based on the IBF flag. Note that the address is eight bits wide with eight DSP16/DSP16A devices (maximum) in the multiprocessor configuration. This means there is one address bit per DSP16/DSP16A device. The srta register has one address bit per device in order to allow transmissions to more than one device at a time. A broadcast mode sending data from one device to all others is accomplished by setting all bits high on the transmission field of srta.

Table 5-3. Time-Division Multiplexed Slot (tdms) Register

Bit	9	8	7	6	5	4	3	2	1	0
Field	SYNCSP	MODE	TRANSMIT SLOT							SYNC

Field	Value	Result/Description
SYNCSP	0	SYNC = ICK/OCK† ÷ 128. ‡
	1	SYNC = ICK/OCK† ÷ 256.
MODE	0	Multiprocessor mode off. DOEN is an input (passive mode).
	1	Multiprocessor mode on. DOEN is an output (active mode).
TRANSMIT SLOT	1xxxxxxx	Transmit slot 7.
	x1xxxxxx	Transmit slot 6.
	xx1xxxxx	Transmit slot 5.
	xxx1xxxx	Transmit slot 4.
	xxxx1xxx	Transmit slot 3.
	xxxxx1xx	Transmit slot 2.
	xxxxxx1x	Transmit slot 1.
SYNC	xxxxxxx1	Transmit slot 0. SYNC is an output (active mode).
	xxxxxxx0	SYNC is an input (passive mode).

† See sioc register, LD field.

‡ Select this mode when in multiprocessor mode.

Typically, the time-division multiplexed slot register (tdms) is set up at the beginning of a program and does not change for each of the devices in the multiprocessor system. If the time slot needs to be changed, it is imperative that each processor still have its own unique time slot. The falling edge of SYNC (the TDM frame sync) is used to update all the new time slots except time slot 0, which is updated in the next cycle of SYNC.

During reset, the tdms register resets to all zeros, disabling the multiprocessor mode by default. The srta register is unaltered by reset.

Table 5-4. Serial Receive/Transmit Address (srta) Register

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Field	RECEIVE ADDRESS								TRANSMIT ADDRESS							

Field	Value	Result/Description
RECEIVE ADDRESS	1xxxxxxx	Receive address 7.
	x1xxxxxx	Receive address 6.
	xx1xxxxx	Receive address 5.
	xxx1xxxx	Receive address 4.
	xxxx1xxx	Receive address 3.
	xxxxx1xx	Receive address 2.
	xxxxxx1x	Receive address 1.
	xxxxxxx1	Receive address 0.
TRANSMIT ADDRESS	1xxxxxxx	Transmit address 7.
	x1xxxxxx	Transmit address 6.
	xx1xxxxx	Transmit address 5.
	xxx1xxxx	Transmit address 4.
	xxxx1xxx	Transmit address 3.
	xxxxx1xx	Transmit address 2.
	xxxxxx1x	Transmit address 1.
	xxxxxxx1	Transmit address 0.

Figure 5-6 shows the operation of a system using eight DSP16/DSP16A devices in a multiprocessor configuration. The settings used for the tdms and srta registers are shown in order to illustrate the current state of these registers during each I/O operation. The following describes the operation shown in Figure 5-6.

Time Slot **Actions**

0 In preparation for time slot 0 (left-most column), the tdms register of device number 7 has been initialized so that it can transmit in time slot zero. This situation also forces the device to generate the frame sync of the I/O stream. The srta register of device 7 has been set so that it can transmit to device 3 and receive address 7. The serial data register (SDX) of device 7 contains the data to be transmitted.

During time slot 0, the data from device 7 is transmitted on the TDM channel. Device 3 recognizes its address on the serial address line (SADD) and accepts the data into its SDX register, which is subsequently read by the command *r0 = sdx. All other devices ignore this transaction, since the transmit address was not theirs.

1 No actions in time slot 1.

2 In preparation for time-slot 2, the tdms register of device 2 has been initialized so that during time slot 2, device 2 will transmit to device 5.

During time slot 2, the data from device 2 is transmitted on the TDM channel. Device 5 recognizes the address on the SADD and accepts the data into its sdx register, which is then read by the command *r1++ = sdx.

3 No actions in time slot 3.

4 No actions in time slot 4.

5 In preparation for time slot 5, device 0 has been initialized so that it will transmit in this time slot to all other devices. Devices 1, 4, and 6, which have not been previously mentioned, are ready to receive data assigned to their respective addresses. Devices 2, 3, 5, and 7, which were initialized earlier, are also ready to receive data.

During time slot 5, the data in device 0 is transmitted on the TDM channel. Every device address is represented on the SADD line, and all devices will accept the data.

6 No actions in time slot 6.

7 No actions in time slot 7.

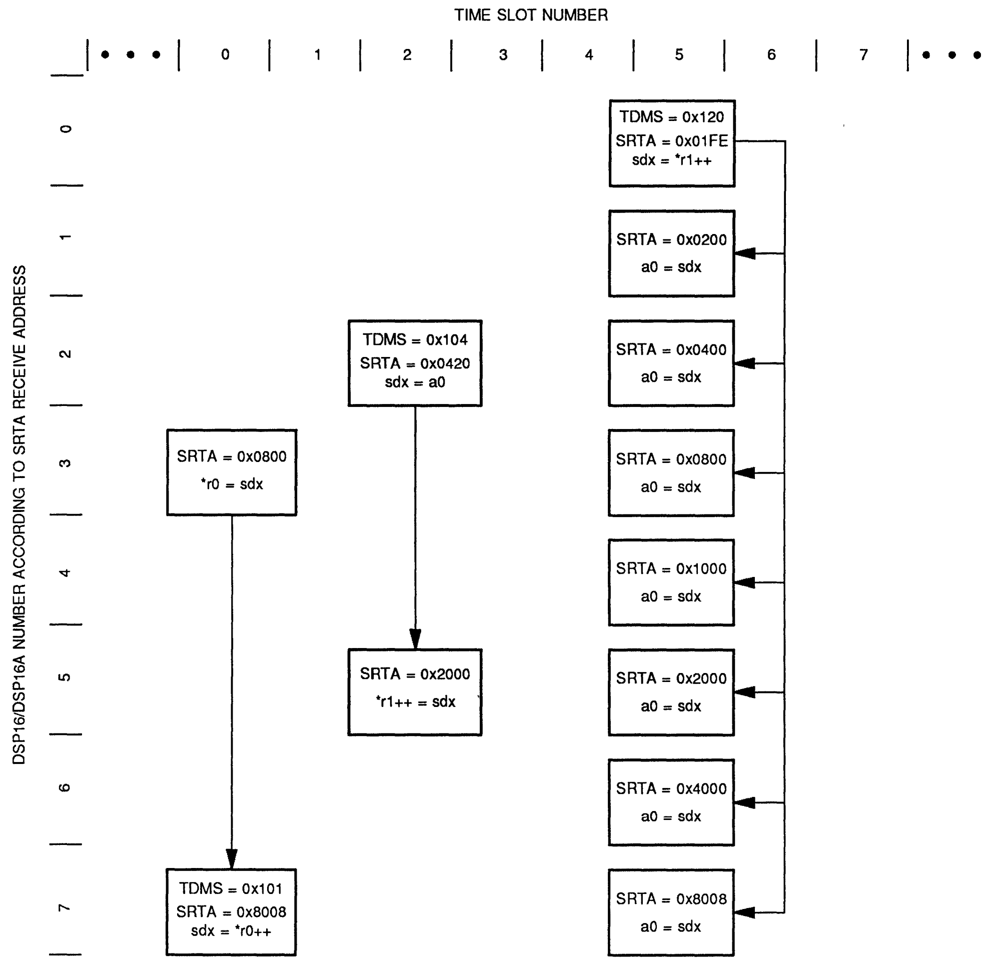


Figure 5-6. DSP16/DSP16A Multiprocessor Communications

5.6.1 Suggested Multiprocessor Configuration

In the suggested configuration, the DSP16/DSP16A device supplying the SYNC signal also supplies the ICK and OCK signals; the remaining DSPs are configured for passive SYNC, ICK, and OCK signals. All DSPs have active ILD and OLD signals.

For the DSP16/DSP16A device with the given transmit slot, the following parameters should be configured as shown:

Parameter	**Transmit Slot 0**	**Transmit Slot 1—7**
SYNC	Active	Passive
ICK	Active	Passive
OCK	Active	Passive
ILD	Active	Active
OLD	Active	Active

To achieve the configuration shown above, the following registers in the DSPs should be set as shown:

Register	**Transmit Slot 0**	**Transmit Slot 1—7**
sioc	0x23C	0x230
tdms	0x101	0x1XX
srta	0xXXX	0xXXX

Note: An "X" indicates that the number is dependent on the specific application.

The interrupt on IBF must also be enabled in the pioc register of each DSP16/DSP16A device to allow the devices to detect and process an input.

5.7 SERIAL I/O TIMING DIAGRAMS

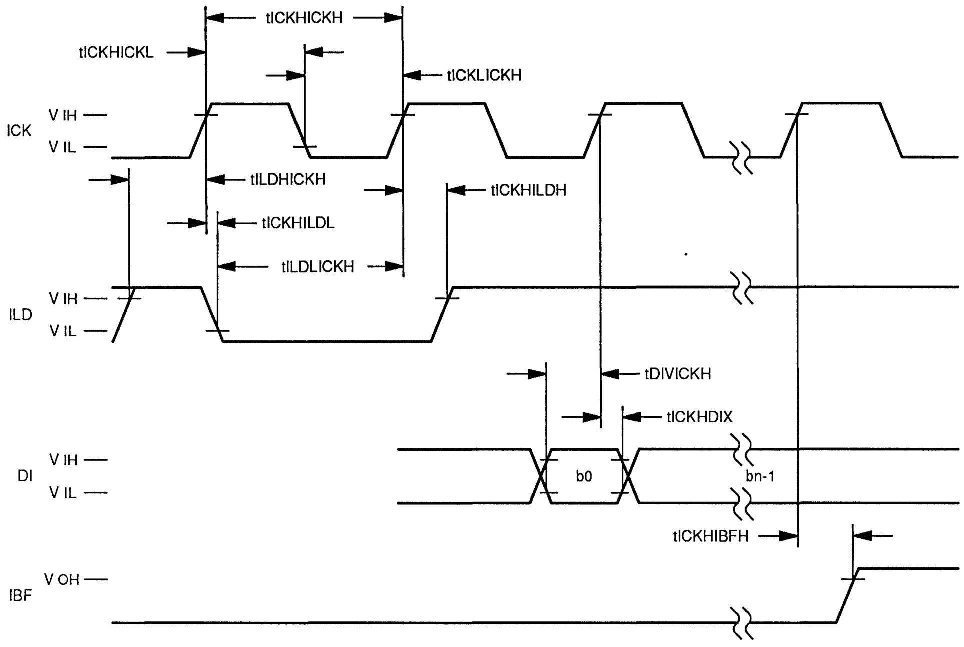


Figure 5-7. Serial Input Timing

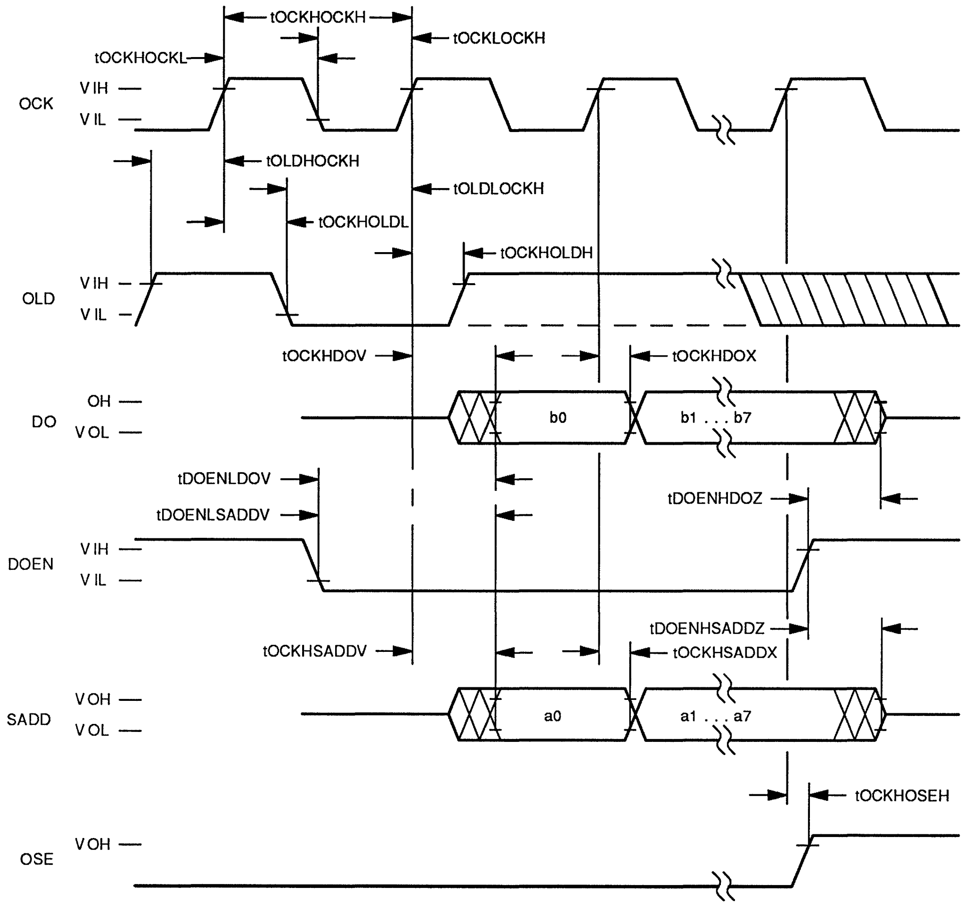


Figure 5-8. Serial Output Timing – 8 Bits

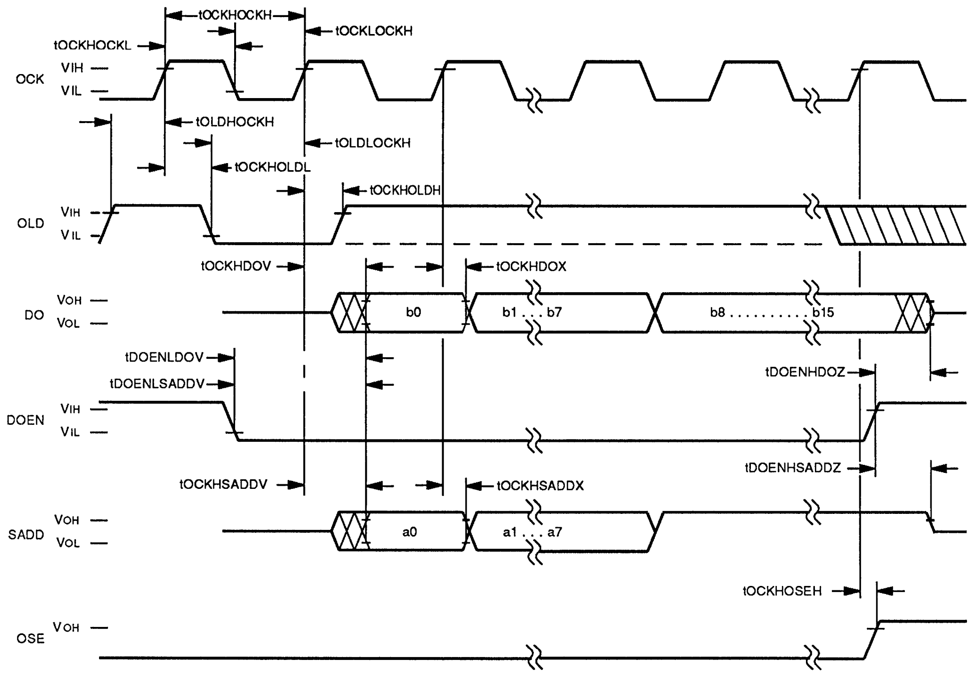


Figure 5-9. Serial Output Timing – 16 Bits

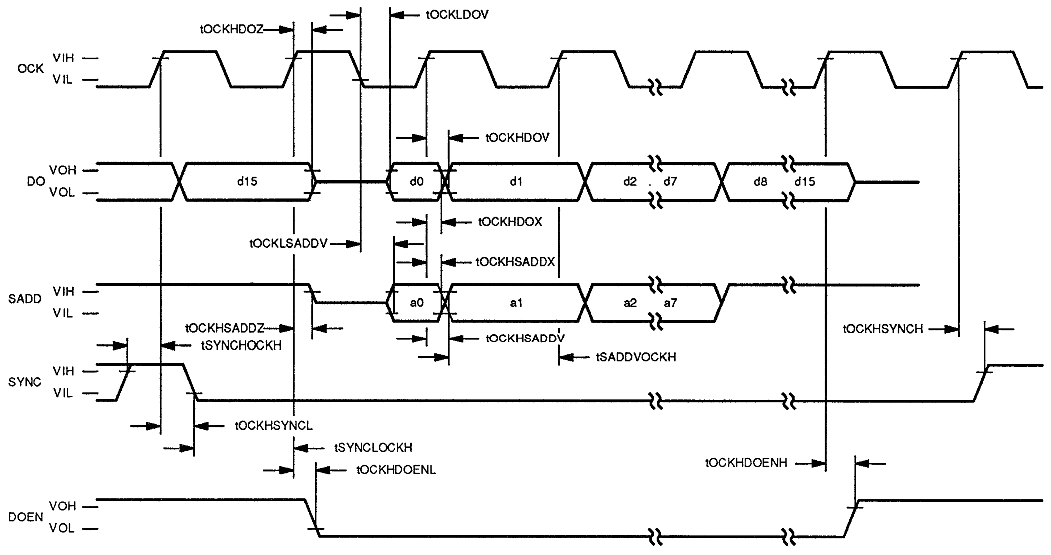


Figure 5-10. Multiprocessor Timing

Chapter 6
Parallel I/O

CHAPTER 6. PARALLEL I/O

CONTENTS

6. PARALLEL I/O

The *WE* DSP16/DSP16A Digital Signal Processor has a 16-bit parallel I/O interface with external devices for rapid transfer of data. Access times are programmable via the strobe field in the pioc register. Minimal or no additional logic is required to interface with memory or other peripheral devices. Five maskable interrupts are included in the PIO unit.

Although there is only one physical PIO port, there are two logical PIO ports: pdx0 and pdx1. The two logical ports are distinguished by the state of the peripheral select line (PSEL). Figure 6-1 shows the DSP16/DSP16A PIO unit at the block level.

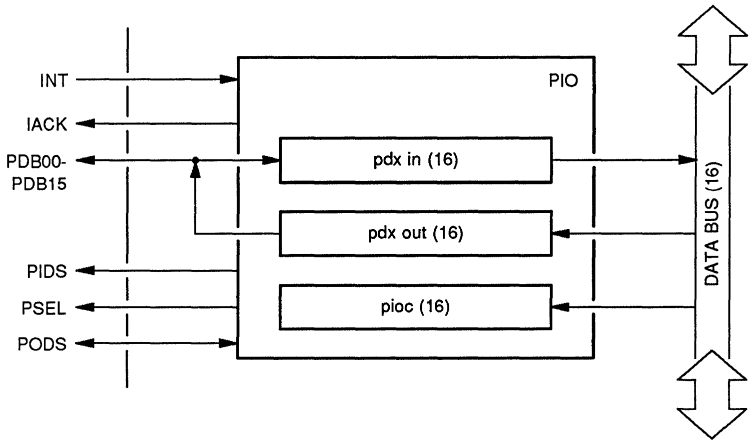


Figure 6-1. Parallel I/O Section

6.1 SOFTWARE DESCRIPTION

The parallel I/O port can be accessed via the data move group of instructions. The two logical ports (pdx0 and pdx1) correspond to the state of the PSEL pin (logical 0 or logical 1, respectively). That is, an access to the logical port pdx0 initiates a transaction with the PSEL signal cleared (0), while an access to logical port pdx1 initiates a transaction with the PSEL signal set (1).

When programming the device, three PIO registers can be referenced:

- pioc – Parallel I/O control register.
- pdx0 – Logical port 0.
- pdx1 – Logical port 1.

Note: pdx0 and pdx1 both reference the same physical register.

The parallel I/O control (pioc) register is a 16-bit, user-accessible register used to configure some features of the parallel I/O:

- External device access time.
- Interrupt masks.
- Active/passive mode.
- Status/control (S/C) bit mode.

Table 6-1 shows the pioc register.

Table 6-1 Parallel I/O Control (pioc) Register

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Field	IBF	STROBE		PODS	PIDS	S/C	INTERRUPTS					STATUS				

Field	Value	Result/Description
IBF	R	IBF interrupt status bit (same as bit 4).
STROBE		Strobe width of **PODS PIDS**
	00	T* T
	01	2T 2T
	10	3T 3T
	11	4T 4T
PODS	0	PODS is an input (passive mode).
	1	PODS is an output (active mode).
PIDS	0	PIDS is an input (passive mode).
	1	PIDS is an output (active mode).
S/C	0	Not S/C mode.
	1	S/C mode.
INTERRUPTS	Wxxxx	IBF interrupt enabled when set.
	xWxxx	OBE interrupt enabled when set.
	xxWxx	PIDS interrupt enabled when set.
	xxxWx	PODS interrupt enabled when set.
	xxxxW	INT interrupt enabled when set.
STATUS	Rxxxx	IBF status bit.
	xRxxx	OBE status bit.
	xxRxx	PIDS status bit.
	xxxRx	PODS status bit.
	xxxxR	INT status bit.

* T = 2 x tCKIHCKIH.

6.1.1 pioc Register Settings

From the system perspective, the DSP16/DSP16A device can be an active device (i.e., initiates transactions on the parallel data bus, PDB), or a passive device (i.e., receives stimulus from external devices). In the active mode, the DSP16/DSP16A device behaves as a bus master; in the passive mode of operation, the DSP16/DSP16A device behaves as a peripheral to another device, such as a microprocessor or another DSP16/DSP16A device. Bit 15 of the pioc register is the same as bit 4. See the following description of bit 4.

Bits 14 and 13 of the **pioc** register control the duration of assertion of the PIDS and PODS signals. This will be described in more detail in Section 6.2.2.

Bit 12 of the pioc register, when equal to logic 1, makes the PODS pin an output; hence, the DSP16/DSP16A device can perform active mode write transactions to external devices. When bit 12 of the pioc register is equal to logic 0, the PODS pin is an input used for passive reads from the DSP16/DSP16A device by external devices.

Bit 11 of the pioc register, when equal to logic 1, makes the PIDS pin an output; hence, the DSP16/DSP16A device can perform active mode read transactions from external devices. When bit 11 of the pioc register is equal to logic 0, the PIDS pin is an input used for passive writes to the DSP16/DSP16A device.

Note: The external PSEL pin always reflects the last access of pdx0 or pdx1 by the DSP16/DSP16A program. PSEL is unaffected by the selection of active or passive PIDS or PODS.

Bit 10 of the pioc register, when equal to logic 1, places the PIO in status and control (S/C) mode. In the S/C mode, the upper half of the parallel data bus (PDB15—PDB08) becomes an input only; the lower half (PDB07—PDB00) remains bidirectional (see Section 6.2.4).

Bits 9, 8, 7, 6, and 5 of the pioc register are used to mask interrupts. There are five types of interrupts that can occur:

- Interrupts caused by an external device writing to the DSP16/DSP16A device's serial port. This type of interrupt is masked when bit 9 of the pioc register is set to logic 0.
- Interrupts caused by an external device reading from the DSP16/DSP16A device's serial port. This type of interrupt is masked when bit 8 of the pioc register is set to logic 0.
- Interrupts caused by an external device writing to the DSP16/DSP16A device's parallel port when the DSP16/DSP16A device is in passive mode. This type of interrupt is masked when bit 7 of the pioc register is set to logic 0.
- Interrupts caused by an external device reading from the DSP16/DSP16A device's parallel port when the DSP16 device is in passive mode. This type of interrupt is masked when bit 6 of the pioc register is set to logic 0.
- Interrupts caused by an external device asserting the INT pin. This type of interrupt is masked when bit 5 of the pioc register is set to logic 0.

Note: There is one instruction latency when altering the INTERRUPTS field of the pioc register. For example, if interrupts are disabled with the command pioc=0x00, the DSP16/DSP16A device still responds to an interrupt during the next instruction. After this instruction has executed, the interrupts are disabled. Therefore, to protect an instruction sequence from interrupts, follow the command to mask the INTERRUPTS field of the pioc register with one instruction that may be safely interrupted.

Bits 4, 3, 2, 1, and 0 of the pioc register indicate the serial and parallel I/O buffers and the interrupt pin. This portion of the pioc register can be used to determine which of the five aforementioned interrupts are requesting service. These bits can be read by an interrupt service routine to determine which interrupt(s) have occurred and, hence, how to proceed to service the interrupt request. These status bits can also be used to perform programmed I/O by polling some condition when necessary. It is important to note that pending interrupt status bits are cleared under the following conditions:

- pioc[4], which indicates that the serial I/O input buffer is full, is cleared by reading from the sdx (serial I/O) register.
- pioc[3], which indicates that the serial output buffer is empty, is cleared when a write to the sdx (serial I/O) register is performed.
- pioc[2], which indicates that an external device has written into the DSP16/DSP16A device's pio register, is cleared when the DSP16/DSP16A device reads from the PIO register (either pdx0 or pdx1). Reading from the PIO register clears this bit irrespective of whether it is done inside or outside an interrupt routine. Note that this interrupt can occur only when the DSP16/DSP16A device is in the passive mode; hence, the DSP16/DSP16A device reading from the PIO registers (to clear pioc[2]) does NOT cause an external read transaction to take place.
- pioc[1], which indicates that an external device has read from the DSP16/DSP16A device's PIO register, is cleared when the DSP16/DSP16A device writes to the PIO register (either pdx0 or pdx1). Writing to the PIO register clears this bit irrespective of whether it is done inside or outside an interrupt routine. Note that this interrupt can occur only when the DSP16/DSP16A device is in the passive mode; hence, writing to the PIO registers (to clear pioc[1]) does NOT cause an external write transaction to take place.
- pioc[0], which indicates that an external device has asserted the INT signal, is cleared when the interrupt acknowledge (IACK) signal makes a high-to-low transition, indicating that the interrupt service routine has completed. If external interrupts are masked, this bit will NOT be set when INT is asserted. Recall that this bit can be cleared only when an ireturn instruction causes the high-to-low transition of IACK.

Note: The contents of the pioc register are cleared, except bit 3 which is set, when the RSTB signal is asserted. Hence, the DSP16/DSP16A device is in passive mode, non-status and control (S/C) bit mode, with all interrupts masked after a chip reset.

6.1.2 Latent Reads

While in active mode, reading from a logical PIO port is accomplished by an actual read of the single physical port on the DSP16/DSP16A device. When a read of the parallel input register (physical port) is performed, a transaction to the external system is performed on the logical port. Reads from the logical port imply that:

- All reads take their data from the on-chip parallel input register.
- As data is read from the internal parallel input register, a read transaction to the external system is initiated.
- Upon completion of the external read transaction, data received from the external system (logical port 0 or 1) is loaded into the parallel input register.

Since data is read from the internal parallel input register and then new data is accepted into the parallel input register from a logical port, reads from the external system are latent. For example, to read a string of four words of data (d0, d1, d2, d3) from the PIO port, the following actions are required:

1. The first instruction reads meaningless data from the parallel input register and initiates the transaction to bring the first datum (d0) from the external device.
2. The second instruction reads the first datum (d0) from the parallel input register and initiates the transaction to bring the second datum (d1) from the external device.
3. The third instruction reads the second datum (d1) from the parallel input register and initiates the transaction to bring the third datum (d2) from the external device.
4. The fourth instruction reads the third datum (d2) from the parallel input register and initiates the transaction to bring the fourth datum (d3) from the external device.
5. The fifth and final instruction reads the fourth datum (d3) from the parallel input register and initiates a transaction that reads another word of data from the external device and overwrites the last datum (d3) in the parallel input register.

To fetch a vector of data of length N requires N+1 instructions and generates N+1 read transactions to the external system. In order to fetch a single datum that is not already present in the parallel input register, two instructions are required. Since both logical ports map into the same physical port, a fetch from either logical port takes data from the parallel input register; the external access then overwrites the contents of the parallel input register with the data from the logical port specified in the instruction.

The parallel output register is distinct from the parallel input register. Writing to pdx0 and pdx1 does not alter the contents of the parallel input register.

6.2 HARDWARE DESCRIPTION

The parallel I/O bus is an asynchronous interface. Data strobes indicate when data may be put on the bus during active mode reads and when data is available on the bus during active mode writes. The PIO port characteristics are programmable and are controlled by the parallel I/O control register (pioc). The following sections describe the PIO signals and their functions.

6.2.1 Parallel I/O Signals

Table 6-2 describes the PIO signals of the DSP16/DSP16A device.

Table 6-2. Parallel I/O Signals

Symbol	Type*	Pin	Name/Description
PDB00	I/O†	11	**Parallel Data Bus – Bit 0.**
PDB01		10	**Parallel Data Bus – Bit 1.**
PDB02		9	**Parallel Data Bus – Bit 2.**
PDB03		8	**Parallel Data Bus – Bit 3.**
PDB04		5	**Parallel Data Bus – Bit 4.**
PDB05		4	**Parallel Data Bus – Bit 5.**
PDB06		3	**Parallel Data Bus – Bit 6.**
PDB07		2	**Parallel Data Bus – Bit 7.**
PDB08		84	**Parallel Data Bus – Bit 8.**
PDB09		83	**Parallel Data Bus – Bit 9.**
PDB10		82	**Parallel Data Bus – Bit 10.**
PDB11		81	**Parallel Data Bus – Bit 11.**
PDB12		78	**Parallel Data Bus – Bit 12.**
PDB13		77	**Parallel Data Bus – Bit 13.**
PDB14		76	**Parallel Data Bus – Bit 14.**
PDB15		75	**Parallel Data Bus – Bit 15.**
PSEL	O†	72	**Peripheral Select.** PSEL is used to specify the logical port to/from which data is to be conveyed. PSEL is high (logic 1) when pdx1 is the register specified in the I/O instruction and low when pdx0 is the register specified.

* I = Input; O = Output.
† 3-stated.

Table 6-2. Parallel I/O Signals (continued)

Symbol	Type*	Pin	Name/Description
PIDS	I/O†	73	**Parallel Input Data Strobe (Active-Low).** In active mode, PIDS is an output. when PIDS is asserted, data may be placed onto the PDB. Upon negation of PIDS, data should be removed from the PDB. PIDS is asserted by the DSP16/DSP16A device during active mode read transaction. In passive mode, PIDS is an input. When asserted by an external device, this signal indicates that data is available on the PDB. In both passive and active modes, the trailing edge (low-to-high transition) of PIDS is the sampling point.
PODS	I/O†	74	**Parallel Output Data Strobe (Active-Low).** In active mode, PODS is an output. When PODS is asserted, data is available on the PDB. PODS is asserted by the DSP16/DSP16A device during an active mode write transaction. In passive mode, PODS is an input. When PODS is asserted by an external device, the DSP16/DSP16A device places the contents of its parallel output register (pdx0 or pdx1) onto the PDB.

* I = Input; O = Output.
† 3-stated.

6.2.2 Active Mode

The duration of parallel I/O strobe signals can be programmed using bits 14 and 13 of the pioc register. Table 6-3 shows the possible configurations.

Table 6-3. PIO Strobe Widths

pioc Bits		Strobe width	
14	13	PIDS*	PODS*
0	0	T	T
0	1	2T	2T
1	0	3T	3T
1	1	4T	4T

* T = 2 x tCKIHCKIH

When minimum strobe widths are configured, parallel I/O transactions can occur at the instruction rate. Consecutive parallel I/O instructions can be executed.

Note: When the strobe widths are not minimum, consecutive PIO instructions are prohibited – other non-PIO instructions must be placed between two PIO instructions.

- When pioc[14, 13] = 01, one or more instructions must be placed between PIO instructions.
- When pioc[14, 13] = 10, two or more instructions must be placed between PIO instructions.
- When pioc[14, 13] = 11, three or more instructions must be placed between PIO instructions.

Any interrupt service routine must guarantee that these conditions are met. As a simple rule, when pioc[14, 13] = 11, the first instruction in an interrupt service routine cannot be a PIO instruction.

6.2.3 Passive Mode

In passive mode, the DSP16/DSP16A device can be used as a peripheral to such other devices as a microprocessor. Bits 12 and 11 of the pioc register are used to configure the passive mode. When bit 12 of the pioc register is clear (0), the PODS signal becomes an input and the contents of the DSP16/DSP16A device's parallel output register can be read by an external device asserting PODS. When bit 11 of the pioc register is clear (0), PIDS is an input and the DSP16/DSP16A device's parallel input register can be written by an external device asserting PIDS.

Providing that their respective interrupt mask bits are set (logic 1), the assertion of PIDS and PODS by an external device causes the DSP16/DSP16A device to recognize an interrupt. This mechanism is a means by which functional synchronization between the DSP16/DSP16A device and an external device can be achieved.

6.2.4 Status and Control (S/C) Mode

The (S/C) mode of the DSP16/DSP16A device's parallel I/O is invoked by setting bit 10 of the pioc register to logic 1. When this bit is set, the upper half of the parallel data bus (PDB15—PDB08) becomes an input only. The lower half of the parallel data bus (PDB07—PDB00) remains bidirectional. The lower half of the parallel data bus can be used as an 8-bit, bidirectional port that behaves just as the 16-bit port does in both the active and passive modes. The upper half of the parallel data bus, being an input only, can perform active and passive mode input transactions.

The S/C bit mode may be used as follows:

- The upper bits (PDB15—PDB08) can be used by devices external to the DSP16/DSP16A device to control the behavior of the DSP16/DSP16A. The DSP16/DSP16A device can be programmed to poll the parallel port and to respond according to the data there. An external device can use the upper eight bits of the parallel I/O (in the passive mode) to write a control byte into a DSP16/DSP16A device. Data on this upper byte must be clocked (using PIDS) into the parallel I/O port. Furthermore, the DSP16/DSP16A device should have its PODS signal configured in the passive mode (bit 12 of pioc clear).

- The lower eight bits of the parallel data bus (PDB07—PDB00) can be used by external devices to indicate some condition (status) internal to the DSP16/DSP16A device. The DSP16/DSP16A device, by writing to the pdx register enables these eight bits when PODS is asserted.

6.3 INTERRUPTS AND THE PARALLEL I/O

There are two internal interrupts generated, based on parallel I/O events. An internal interrupt is generated (provided it is unmasked) when an external device performs a passive mode write. When the external device negates the PIDS signal (low-to-high transition), an internal interrupt request is generated. When the DSP16/DSP16A device accepts this interrupt request, the IACK signal is asserted. When the interrupt routine is completed, IACK is negated (becomes logic 0). See Section 2.8 for more information on how the DSP16/DSP16A device reacts to interrupts.

Similarly, when an external device performs a passive read, an internal interrupt request is generated upon negation (low-to-high transition) of PODS. When the DSP16/DSP16A device accepts this interrupt request, the IACK signal is asserted. When the interrupt routine has completed, IACK is negated (becomes logic 0).

When the DSP16/DSP16A device is in the passive mode, program synchronization can be obtained by using the interrupt mechanism to synchronize a data source, with the program being run by the DSP16/DSP16A device. Briefly, the DSP16/DSP16A device can have its parallel I/O configured in the passive mode. A data source provides data to the DSP16/DSP16A device via passive writes. During the associated interrupt routine, the DSP16/DSP16A device program performs I/O functions. The receipt of data by the DSP16/DSP16A device is indicated to the external data source by the falling edge (high-to-low) transition of the IACK signal.

When the PIDS signal is active, the **pdx in** register is shadowed during interrupts. This allows the parallel input to be used during interrupts without the possibility of destroying data previously fetched via a latent PIO read. When the interrupt service routine is exited, **pdx in** is loaded with its value prior to the interrupt. If the parallel input is changed from active to passive during the interrupt, the shadowing feature is disabled.

6.4 PIO BUS TRANSACTIONS

In this section, the four types of parallel I/O transactions are described:

- Active mode input (active read)
- Active mode output (active write)
- Passive mode input (passive read)
- Passive mode output (passive write)

Note: For timing information, refer to the appropriate data sheet.

6.4.1 Active Mode Input

The active mode input transaction (shown in Figure 6-2) is initiated by the DSP16/DSP16A device executing a data move instruction (e.g., *r2 = pdx0). When an active mode input occurs, PIDS and PSEL are asserted, indicating that an external device can place data on the parallel data bus (PDB). PSEL can be used to select one of two external sources for the data. The external device must place valid data on the PDB before the negation of PIDS. The access time for the external data source is configurable via the pioc register. The external data source can remove data from the PDB after negation of PIDS.

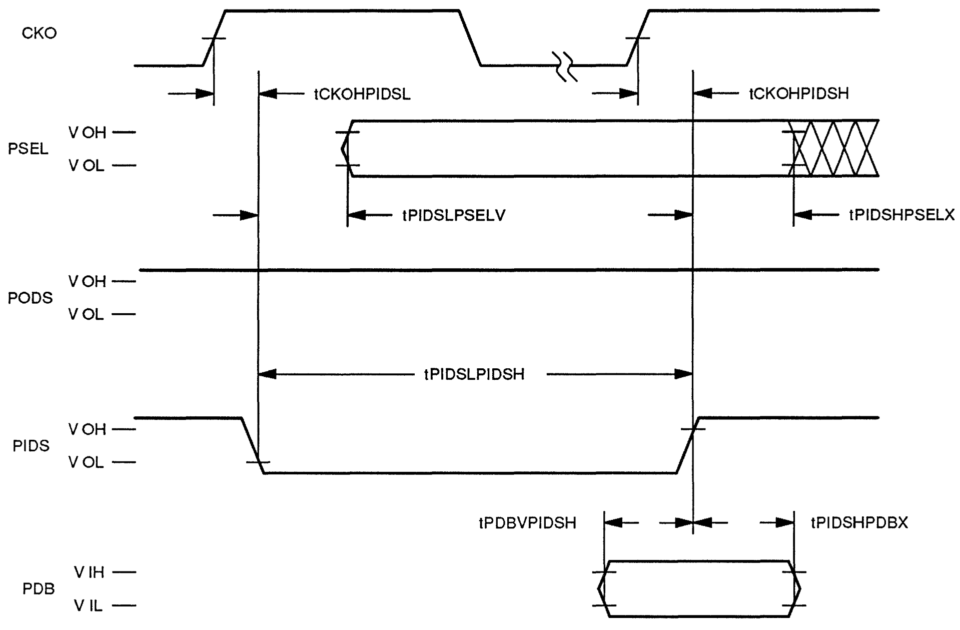


Figure 6-2. Active Mode Input Timing

6.4.2 Active Mode Output

The active mode output transaction (shown in Figure 6-3) is initiated by the DSP16/DSP16A device executing a data move instruction (e.g., pdx0 = *r2). When an active mode output occurs, PODS and PSEL are asserted. A short time later, the DSP16/DSP16A device places data onto the PDB. This data remains valid until a short time after the negation of PODS. This write time is configurable via the pioc register.

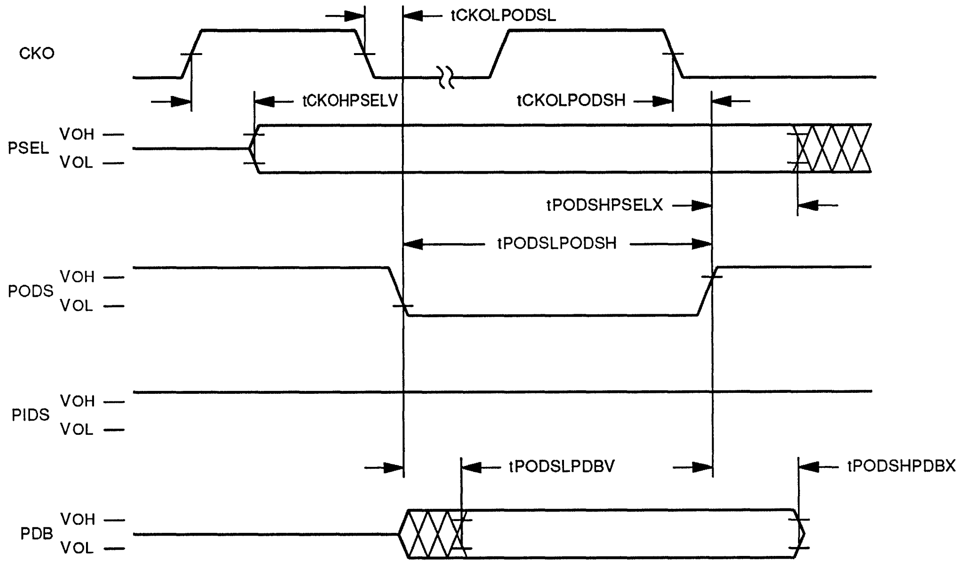


Figure 6-3. Active Mode Output Timing

6.4.3 Passive Mode Input

The passive mode input transaction (shown in Figure 6-4) is initiated by an external device asserting PIDS. The external device must place data onto the PDB prior to the negation of PIDS. The external device may remove the data from the bus after the negation of PIDS.

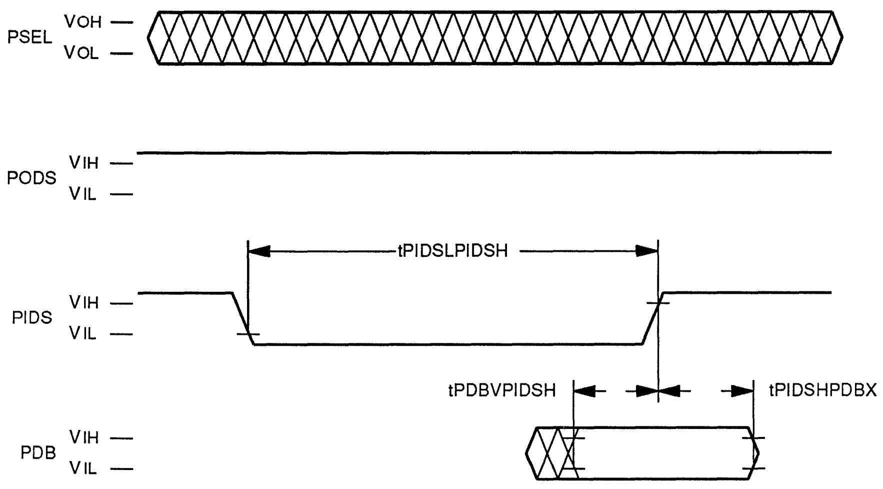


Figure 6-4. Passive Mode Input Timing

6.4.4 Passive Mode Output

The passive mode output transaction (shown in Figure 6-5) is initiated by an external device asserting PODS. A short time later, the DSP16/DSP16A device places data from its pdx output register onto the PDB. The data remains valid until a short time after the negation of PODS by the external device.

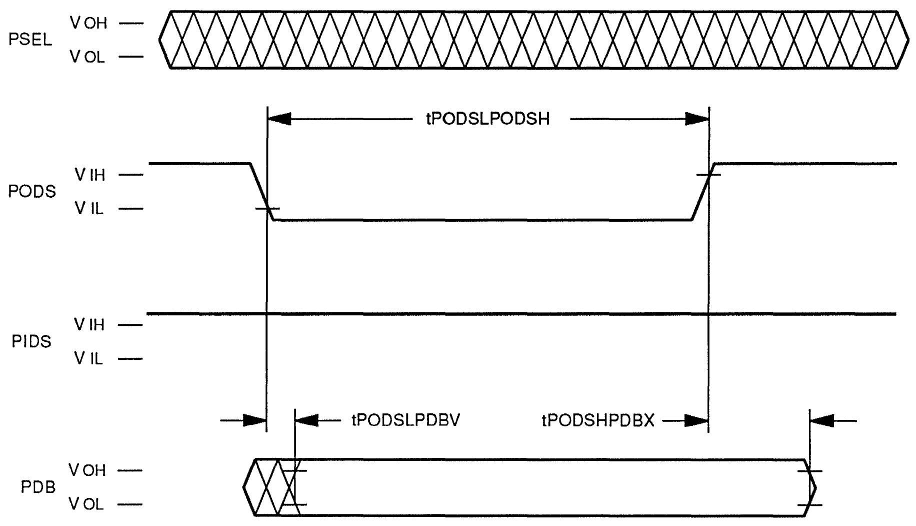


Figure 6-5. Passive Mode Output Timing

6.4.5 Parallel I/O Interaccess Timing

Figure 6-6 shows the timing of mixed active mode inputs and outputs. Refer to the previous sections (6.4.1 and 6.4.2) on active mode input and output transactions for specific descriptions of individual transactions.

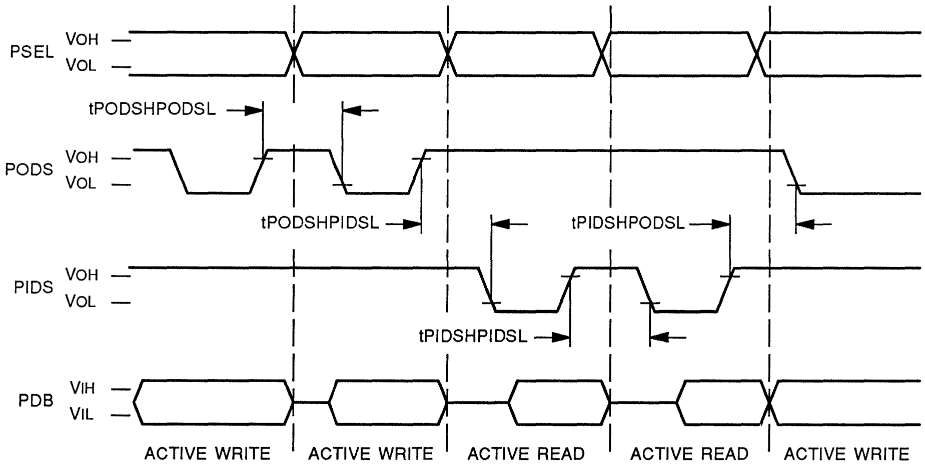


Figure 6-6. Parallel I/O Interaccess Timing

Chapter 7
Interface Guide

CHAPTER 7. INTERFACE GUIDE

CONTENTS

7. INTERFACE GUIDE

This chapter describes elements of the physical design of the DSP16 device that must be considered when designing practical physical systems. For timing characteristics and specifications, refer to an up-to-date *WE® DSP16 Digital Signal Processor* Data Sheet or *WE® DSP16A Digital Signal Processor* Data Sheet. For information concerning the 133-pin PGA package, refer to the *WE® DSP16 Digital Signal Processor For Military Applications* Data Sheet.

Note: When working with critical information, be sure it is the latest available. The date of publication can be found on the last page of data sheets and on the title page of manuals. Normally, data sheets are updated more frequently than information manuals.

7.1 PIN ASSIGNMENTS

Figure 7-1 is the pin diagram for the DSP16/DSP16A device. Table 7-1 is an alphabetical list of pin names.

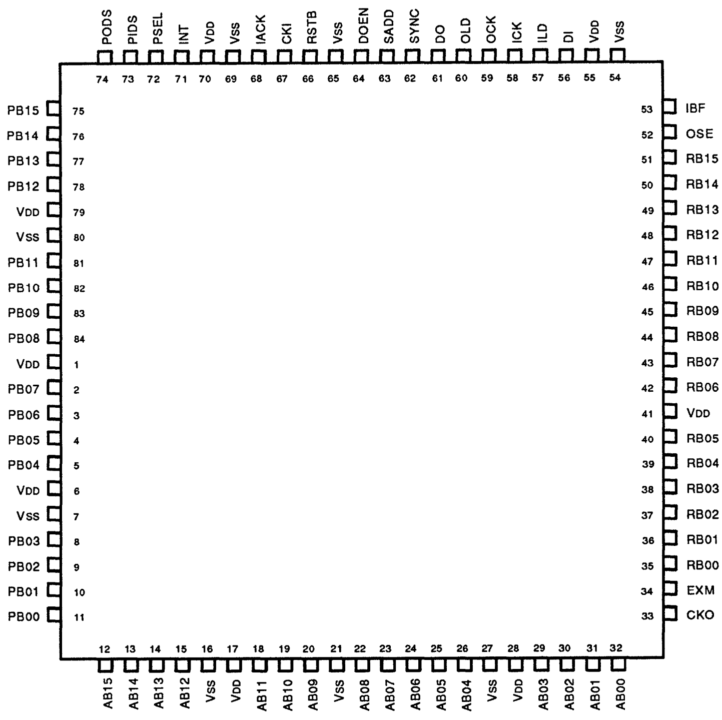


Figure 7-1. DSP16/DSP16A Digital Signal Processor – Pin Diagram

Table 7-1. DSP16/DSP16A Pin Names

Symbol	Pin	Symbol	Pin
AB00—AB15	32—29, 26—22, 20—18, 15—12	OLD	60
		OSE	52
CKI	67	PB00—PB15	11—8, 5—2 84—81, 78—75
CKO	33		
DI	56		
DO	61	PIDS	73
DOEN	64	PODS	74
EXM	34	PSEL	72
IACK	68	RB00—RB15	35—40, 42—51
IBF	53	RSTB	66
ICK	58	SADD	63
ILD	57	SYNC	62
INT	71	VDD	1, 6, 17, 28, 41, 55, 70, 79
OCK	59		
		VSS	7, 16, 21, 27, 54, 65, 69, 80

7.1.1 Device Pins by Numerical Order

Table 7-2. DSP16/DSP16A Pin Descriptions

Pin	Symbol	Type*	Name/Description
1	VDD	P	5 V.
2	PB07	I/O†	Parallel I/O Data Bus – Bit 7.
3	PB06	I/O†	Parallel I/O Data Bus – Bit 6.
4	PB05	I/O†	Parallel I/O Data Bus – Bit 5.
5	PB04	I/O†	Parallel I/O Data Bus – Bit 4.
6	VDD	P	5 V.
7	VSS	P	Ground.
8	PB03	I/O†	Parallel I/O Data Bus – Bit 3.
9	PB02	I/O†	Parallel I/O Data Bus – Bit 2.
10	PB01	I/O†	Parallel I/O Data Bus – Bit 1.
11	PB00	I/O†	Parallel I/O Data Bus – Bit 0.
12	AB15	O†	ROM Address Bus – Bit 15.
13	AB14	O†	ROM Address Bus – Bit 14.
14	AB13	O†	ROM Address Bus – Bit 13.
15	AB12	O†	ROM Address Bus – Bit 12.

* I = Input; O = Output; P = Power Supply.
† 3-stated.

Table 7-2. DSP16/DSP16A Pin Descriptions (Continued)

Pin	Symbol	Type*	Name/Description
16	Vss	P	**Ground.**
17	VDD	P	**5 V.**
18	AB11	O†	**ROM Address Bus – Bit 11.**
19	AB10	O†	**ROM Address Bus – Bit 10.**
20	AB09	O†	**ROM Address Bus – Bit 9.**
21	Vss	P	**Ground.**
22	AB08	O†	**ROM Address Bus – Bit 8.**
23	AB07	O†	**ROM Address Bus – Bit 7.**
24	AB06	O†	**ROM Address Bus – Bit 6.**
25	AB05	O†	**ROM Address Bus – Bit 5.**
26	AB04	O†	**ROM Address Bus – Bit 4.**
27	Vss	P	**Ground.**
28	VDD	P	**5 V.**
29	AB03	O†	**ROM Address Bus – Bit 3.**
30	AB02	O†	**ROM Address Bus – Bit 2.**
31	AB01	O†	**ROM Address Bus – Bit 1.**
32	AB00	O†	**ROM Address Bus – Bit 0.**
33	CKO	O	**Clock Out.** Buffered clock at half the frequency of CKI.
34	EXM	I	**External Memory.** When high, all instructions and coefficients are fetched from external memory. When low: **DSP16** – forces use of internal ROM for instructions and coefficients. **DSP16A** – the first 4 Kwords of program memory are fetched from internal ROM; addresses beyond 4 Kwords are fetched from external memory.
35	RB00	I	**ROM Data Bus – Bit 0.**
36	RB01	I	**ROM Data Bus – Bit 1.**
37	RB02	I	**ROM Data Bus – Bit 2.**
38	RB03	I	**ROM Data Bus – Bit 3.**
39	RB04	I	**ROM Data Bus – Bit 4.**
40	RB05	I	**ROM Data Bus – Bit 5.**
41	VDD	P	**5 V.**
42	RB06	I	**ROM Data Bus – Bit 6.**
43	RB07	I	**ROM Data Bus – Bit 7.**
44	RB08	I	**ROM Data Bus – Bit 8.**
45	RB09	I	**ROM Data Bus – Bit 9.**
46	RB10	I	**ROM Data Bus – Bit 10.**

* I = Input; O = Output; P = Power Supply.
† 3-stated.

Table 7-2. DSP16/DSP16A Pin Descriptions (Continued)

Pin	Symbol	Type*	Name/Description
47	RB11	I	**ROM Data Bus – Bit 11.**
48	RB12	I	**ROM Data Bus – Bit 12.**
49	RB13	I	**ROM Data Bus – Bit 13.**
50	RB14	I	**ROM Data Bus – Bit 14.**
51	RB15	I	**ROM Data Bus – Bit 15.**
52	OSE	O†	**Output Shift Register Empty**. Indicates the end of a serial transmission. OSE is set either by emptying the output shift register or by asserting RSTB. OSE is reset by the DSP16 device writing a word to the output shift register.
53	IBF	O†	**Input Buffer Full.** IBF is set when the input buffer is filled and cleared by a read of the buffer. IBF is also cleared by asserting RSTB.
54	VSS	P	**Ground.**
55	VDD	P	**5 V.**
56	DI	I	Serial PCM data latched on rising edge of ICK, either LSB or MSB first, according to the status of the sioc register MSB field.
57	ILD	I/O†	**Input Load**. Falling edge of ILD indicates the beginning of a serial input word. In active mode, ILD is an output; in passive mode, ILD is an input, depending on the sioc register ILD field.
58	ICK	I/O†	**Input Clock**. Clock for serial PCM input data. In active mode, ICK is an output; in passive mode, ICK is an input, depending on the I/O format.
59	OCK	I/O†	**Output Clock**. Clock for serial PCM output data. In active mode, OCK is an output; in passive mode, OCK is an input, depending on the I/O format.
60	OLD	I/O†	**Output Load**. Clock for loading the parallel-to-serial converter from the output buffer (obuf). A falling edge of OLD indicates the beginning of a serial output word. In active mode, OLD is an output; in passive, OLD is an input, according to the sioc register OLD field.

* I = Input; O = Output; P = Power Supply.
† 3-stated.

Table 7-2. DSP16/DSP16A Pin Descriptions (Continued)

Pin	Symbol	Type*	Name/Description
61	DO	O†	**Data Output**. Serial PCM data output from the output shift (osr) register, either LSB or MSB first, according to the sioc register MSB field. DO changes on the rising edges of OCK.
62	SYNC	I/O†	**Multiprocessor Synchronization**. A falling edge of SYNC indicates the first word of a TDM I/O stream. SYNC is an output when the tdms register transmit slot 0 is set; otherwise, it is input.
63	SADD	I/O†	**Multiprocessor Address (Active-Low)**. An 8-bit serial bit stream used for addressing during multiprocessor communication between multiple DSP16 devices. SADD is an output when the tdms time slot dictates a serial transmission; otherwise, it is an input.
64	DOEN	I/O†	**Data Output Enable**. An output when in the multiprocessor mode (tdms register MODE field set) and an input otherwise. DO is 3-stated when DOEN is high.
65	Vss	P	**Ground.**
66	RSTB	I	**Reset**. A high-to-low transition causes entry into the reset state. The sioc, pioc (except bit 3, which is set), tdms, rb, and re register bits are cleared. Reset clears external flags IACK and IBF and sets external flag OSE. DAU condition flags and the auc register are not affected by reset. All output and bidirectional pins are 3-stated during reset. A low-to-high transition causes execution to begin at ROM location 0.
67	CKI	I	**Clock In**. Input clock at twice the frequency of internal operations.
68	IACK	O†	**Interrupt Acknowledge**. IACK signals when an interrupt is being serviced by the DSP16. The IACK remains high until normal instruction operation resumes.
69	Vss	P	**Ground.**
70	VDD	P	**5 V.**

* I = Input; O = Output; P = Power Supply.
† 3-stated.

Table 7-2. DSP16/DSP16A Pin Descriptions (Continued)

Pin	Symbol	Type*	Name/Description
71	INT	I	**Processor Interrupt**. Interrupt to DSP16. INT is acknowledged when interrupts are enabled by the pioc register.
72	PSEL	O†	**Peripheral Select**. PSEL is used in the input and output modes to address external devices. PSEL is low when register pdx0 is referenced, and high when pdx1 is referenced.
73	PIDS	I/O†	**Parallel Input Data Strobe**. Active mode (output) is controlled by data move instructions. Passive mode (input) is externally controlled. On low-to-high transition, data from the parallel I/O data bus is latched into the parallel input data register.
74	PODS	I/O†	**Parallel Output Data Strobe**. Active mode (output) is controlled by data move instructions. Passive mode (input) is externally controlled. When low, data from the parallel data output register is enabled into the parallel I/O data bus. The rising edge may be used as a latching clock. When high, the parallel I/O data bus is 3-stated.
75	PB15	I/O†	**Parallel I/O Data Bus – Bit 15**
76	PB14	I/O†	**Parallel I/O Data Bus – Bit 14**
77	PB13	I/O†	**Parallel I/O Data Bus – Bit 13**
78	PB12	I/O†	**Parallel I/O Data Bus – Bit 12**
79	VDD	P	**5 V.**
80	VSS	P	**Ground**
81	PB11	I/O†	**Parallel I/O Data Bus – Bit 11**
82	PB10	I/O†	**Parallel I/O Data Bus – Bit 10**
83	PB09	I/O†	**Parallel I/O Data Bus – Bit 9**
84	PB08	I/O†	**Parallel I/O Data Bus – Bit 8**

* I = Input; O = Output; P = Power Supply.
† 3-stated.

7.1.2 Pins by Functional Group

Table 7-3. External Memory Interface Group

Pin	Symbol	Type*	Name/Description
32	AB00		**ROM Address Bus – Bit 0.**
31	AB01		**ROM Address Bus – Bit 1.**
30	AB02		**ROM Address Bus – Bit 2.**
29	AB03		**ROM Address Bus – Bit 3.**
26	AB04		**ROM Address Bus – Bit 4.**
25	AB05		**ROM Address Bus – Bit 5.**
24	AB06		**ROM Address Bus – Bit 6.**
23	AB07		**ROM Address Bus – Bit 7.**
22	AB08	O†	**ROM Address Bus – Bit 8.**
20	AB09		**ROM Address Bus – Bit 9.**
19	AB10		**ROM Address Bus – Bit 10.**
18	AB11		**ROM Address Bus – Bit 11.**
15	AB12		**ROM Address Bus – Bit 12.**
14	AB13		**ROM Address Bus – Bit 13.**
13	AB14		**ROM Address Bus – Bit 14.**
12	AB15		**ROM Address Bus – Bit 15.**
34	EXM	I	**External Memory.** When high, all instructions and coefficients are fetched from external memory. When low: **DSP16** – forces use of internal ROM for instructions and coefficients. **DSP16A** – the first 4 Kwords of program memory are fetched from internal ROM; addresses beyond 4 Kwords are fetched from external memory.
35	RB00		**ROM Data Bus – Bit 0.**
36	RB01		**ROM Data Bus – Bit 1.**
37	RB02		**ROM Data Bus – Bit 2.**
38	RB03		**ROM Data Bus – Bit 3.**
39	RB04		**ROM Data Bus – Bit 4.**
40	RB05		**ROM Data Bus – Bit 5.**
42	RB06		**ROM Data Bus – Bit 6.**
43	RB07	I	**ROM Data Bus – Bit 7.**
44	RB08		**ROM Data Bus – Bit 8.**
45	RB09		**ROM Data Bus – Bit 9.**
46	RB10		**ROM Data Bus – Bit 10.**
47	RB11		**ROM Data Bus – Bit 11.**
48	RB12		**ROM Data Bus – Bit 12.**
49	RB13		**ROM Data Bus – Bit 13.**
50	RB14		**ROM Data Bus – Bit 14.**
51	RB15		**ROM Data Bus – Bit 15.**

* I = Input; O = Output
† 3-stated.

Table 7-4. SIO Interface Group

Pin	Symbol	Type*	Name/Description
52	OSE	O†	**Output Shift Register Empty**. Indicates the end of a serial transmission. OSE is set either by emptying the output shift register or by asserting RSTB. OSE is reset by the DSP16 writing a word to the output shift register.
53	IBF	O†	**Input Buffer Full**. IBF is set when the input buffer is filled and cleared by a read of the buffer. IBF is also cleared by asserting RSTB.
56	DI	I	**Data Input**. Serial PCM data latched on rising edge of ICK, either LSB or MSB first, according to the status of the sioc register MSB field.
57	ILD	I/O†	**Input Load**. Falling edge of ILD indicates the beginning of a serial input word. In active mode, ILD is an output; in passive mode, ILD is an input, depending on the sioc register ILD field.
58	ICK	I/O†	**Input Clock**. Clock for serial PCM input data. In active mode, ICK is an output; in passive mode, ICK is an input, depending on the I/O format.
59	OCK	I/O†	**Output Clock**. Clock for serial PCM output data. In active mode, OCK is an output; in passive mode, OCK is an input, depending on the I/O format.
60	OLD	I/O†	**Output Load**. Clock for loading the parallel-to-serial converter from the output buffer (obuf). A falling edge of OLD indicates the beginning of a serial output word. In active mode, OLD is an output; in passive mode, OLD is an input, according to the sioc register OLD field.
61	DO	O†	**Data Output**. Serial PCM data output from the output shift (osr) register, either LSB or MSB first, according to the sioc register MSB field. DO changes on the rising edges of OCK.

* I = Input; O = Output
† 3-stated.

Table 7-4. SIO Interface Group (Continued)

Pin	Symbol	Type*	Name/Description
62	SYNC	I/O†	**Multiprocessor Synchronization**. A falling edge of SYNC indicates the first word of a TDM I/O stream. SYNC is an output when the tdms register transmit slot 0 is set; otherwise, it is an input.
63	SADD	I/O†	**Multiprocessor Address (Active-Low)**. An 8-bit serial bit stream used for addressing during multiprocessor communication between multiple DSP16 devices. SADD is an output when the tdms time slot dictates a serial transmission; otherwise, it is an input.
64	DOEN	I/O†	**Data Output Enable**. An output when in the multiprocessor mode (tdms register MODE field set) and an input otherwise. DO is 3-stated when DOEN is high.

Table 7-5. PIO Interface Group

Pin	Symbol	Type*	Name/Description
11	PDB00	I/O†	**Parallel I/O Data Bus – Bit 0.**
10	PDB01	I/O†	**Parallel I/O Data Bus – Bit 1.**
9	PDB02	I/O†	**Parallel I/O Data Bus – Bit 2.**
8	PDB03	I/O†	**Parallel I/O Data Bus – Bit 3.**
5	PDB04	I/O†	**Parallel I/O Data Bus – Bit 4.**
4	PDB05	I/O†	**Parallel I/O Data Bus – Bit 5.**
3	PDB06	I/O†	**Parallel I/O Data Bus – Bit 6.**
2	PDB07	I/O†	**Parallel I/O Data Bus – Bit 7.**
84	PDB08	I/O†	**Parallel I/O Data Bus – Bit 8.**
83	PDB09	I/O†	**Parallel I/O Data Bus – Bit 9.**
82	PDB10	I/O†	**Parallel I/O Data Bus – Bit 10.**
81	PDB11	I/O†	**Parallel I/O Data Bus – Bit 11.**
78	PDB12	I/O†	**Parallel I/O Data Bus – Bit 12.**
77	PDB13	I/O†	**Parallel I/O Data Bus – Bit 13.**
76	PDB14	I/O†	**Parallel I/O Data Bus – Bit 14.**
75	PDB15	I/O†	**Parallel I/O Data Bus – Bit 15.**

* I = Input; O = Output
† 3-stated.

Table 7-5. PIO Interface Group (Continued)

Pin	Symbol	Type*	Name/Description
72	PSEL	O†	**Parallel Select**. PSEL is used to address external devices. PSEL is low when pdx0 is referenced and high when pdx1 is referenced.
73	PIDS	I/O†	**Parallel Input Data Strobe**. Active mode (output) is controlled by the data move instructions. Passive mode (input) is externally controlled. On low-to-high transition, data from the parallel I/O data bus is latched into the parallel input data register.
74	PODS	I/O†	**Parallel Output Data Strobe**. Active mode (output) is controlled by the data move instructions. Passive mode (input) is externally controlled. When low, data from the parallel data output register is placed into the parallel data bus. The rising edge may be used as a latching clock. When high, the parallel I/O data bus is 3-stated.

Table 7-6. Miscellaneous Function Group

Pin	Symbol	Type*	Name/Description
33	CKO	O†	**Clock Out**. Buffered clock at half the frequency of CKI.
66	RSTB	I	**Reset**. A high-to-low transition causes entry into the reset state. The sioc, pioc (except bit 3, which is set), tdms, rb, and re register bits are cleared. Reset clears external flags IACK and IBF and sets external flag OSE. DAU condition flags and the auc register are not affected by reset. All output and bidirectional pins are 3-stated during reset. A low-to-high transition causes execution to begin at ROM location 0.
67	CKI	I	**Clock In**. Input clock at twice the frequency of internal operations.
68	IACK	O†	**Interrupt Acknowledge**. Interrupt acknowledge signals when an interrupt is being serviced by the DSP16 device. The IACK remains high until normal instruction operation resumes.
71	INT	I	**Processor Interrupt**. Interrupt to DSP16 device. INT is acknowledged when interrupts are enabled by the pioc register.

* I = Input; O = Output
† 3-stated.

Table 7-7. Power and Ground Group

Pin	Symbol	Type*	Name/Description
1	VDD		5 V.
6	VDD		5 V.
17	VDD		5 V.
28	VDD	P	5 V.
41	VDD		5 V.
55	VDD		5 V.
70	VDD		5 V.
79	VDD		5 V.
7	VSS		Ground.
16	VSS		Ground.
21	VSS		Ground.
27	VSS	P	Ground.
54	VSS		Ground.
65	VSS		Ground.
69	VSS		Ground.
80	VSS		Ground.

* P = Power Supply.

7.2 ELECTRICAL CHARACTERISTICS

The parameters are valid for the following conditions: TC = 0 to 85 °C;
VDD = 5 V ± 10%; VSS = 0 V; t = 2 × tCKIHCKIH.

Table 7-8. Electrical Characteristics

Parameter	Sym	Min	Max	Unit
Input Voltage				
Low	VIL	—	0.8	V
High	VIH	2.0	—	V
Output Voltage				
Low (IOL=2.0 mA)	VOL	—	0.4	V
High (IOH= –2.0 mA)	VOH	2.4	—	V
Output Current				
Low (IOL=0.4 V)	IOL	—	2.0	mA
High (IOH=2.4 V)	IOH	—	–2.0	mA
Output Short-Circuit Current (VOH=0 V)	IOS	—	–200	mA
Output 3-State Current				
Low (VIL = 0.8)	IOZL	–75	75	µA
High (VIH = 2.0)	IOZH	–75	75	µA
Input Current				
Low (VIL = 0.8)	IIL	—	–25	µA
High (VIH = 2.0)	IIH	—	25	µA
Input Capacitance	CI	—	15	pF

Absolute Maximum Ratings

Voltage on any pin with respect to ground	–0.5 V to +6 V
Power dissipation	1 W
Ambient temperature range	–40 °C to +120 °C
Storage temperature range	–65 °C to +150 °C

Maximum ratings are the limiting conditions that can be applied to all variations of circuit and environmental conditions without the occurrence of permanent damage.

Bonding and soldering of the external leads of these devices can be performed safely at temperatures up to 300 °C.

7.3 EXTERNAL MEMORY

Figure 7-2 shows the external memory interface timing. Note that there are no read or write signals. The external memory interface was primarily provided to allow users to place programs in external ROM, rather than having to order mask-programmed devices. In some applications, such as the DSP16 and DSP16A Development Systems, the external memory is RAM, which is loaded by a microcomputer and then read by the DSP16/DSP16A device.

DSP16 Only

When the EXM pin is tied high, the internal ROM of the device can be replaced with up to 64 Kwords of external memory. The address space starts at address zero. Addresses are generated by the XAAU (ROM addressing unit) and supplied off-chip on the address bus (AB00—AB15).

DSP16A Only

When the EXM pin is tied low, the processor augments the low 4 Kwords of on-chip program ROM with up to 60 Kwords of external program memory. When the EXM pin is tied high, the entire 64 Kword address space is mapped into external program memory.

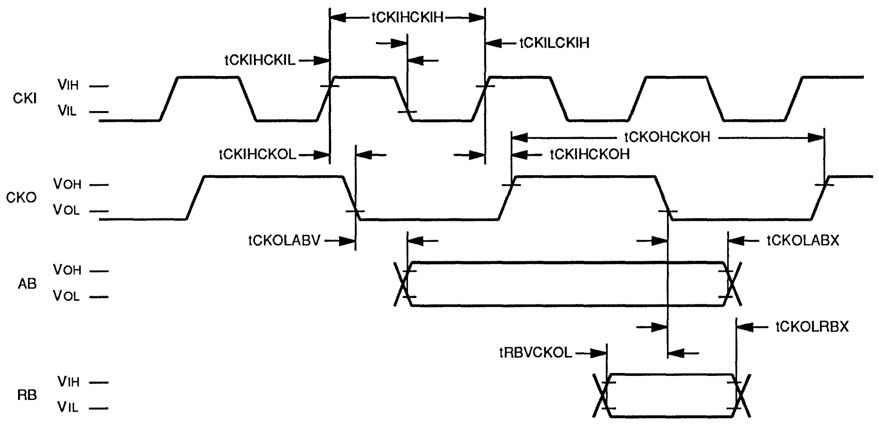
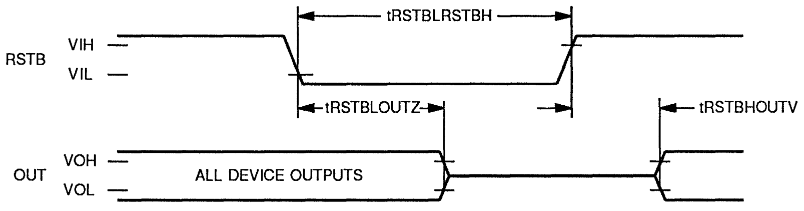


Figure 7-2. External Memory Interface Timing

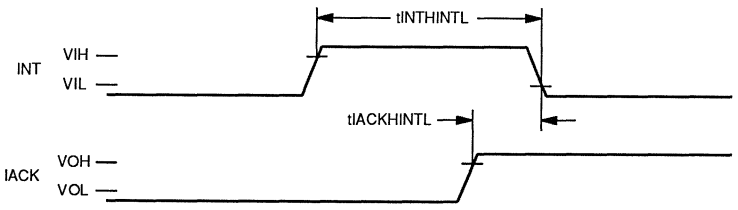
7.4 RESET AND INTERRUPT CONTROL

The DSP16/DSP16A device can be reset by the use of the RSTB input. Asserting RSTB (low) causes all device outputs to 3-state. The next low-to-high transition causes the device to begin program execution from address zero. Part A of Figure 7-3 shows the timing of the RSTB input and device outputs.

Part B of Figure 7-3 shows the timing of the INT (interrupt request) input. When an external device requests an interrupt, the DSP16 device recognizes an interrupting condition at the next interruptible instruction (provided the INT interrupt enable bit in the pioc register is set), the interrupt acknowledge signal (IACK) is asserted, and program execution branches to address one. At the end of the interrupt service routine, as determined by the ireturn instruction, IACK is negated.



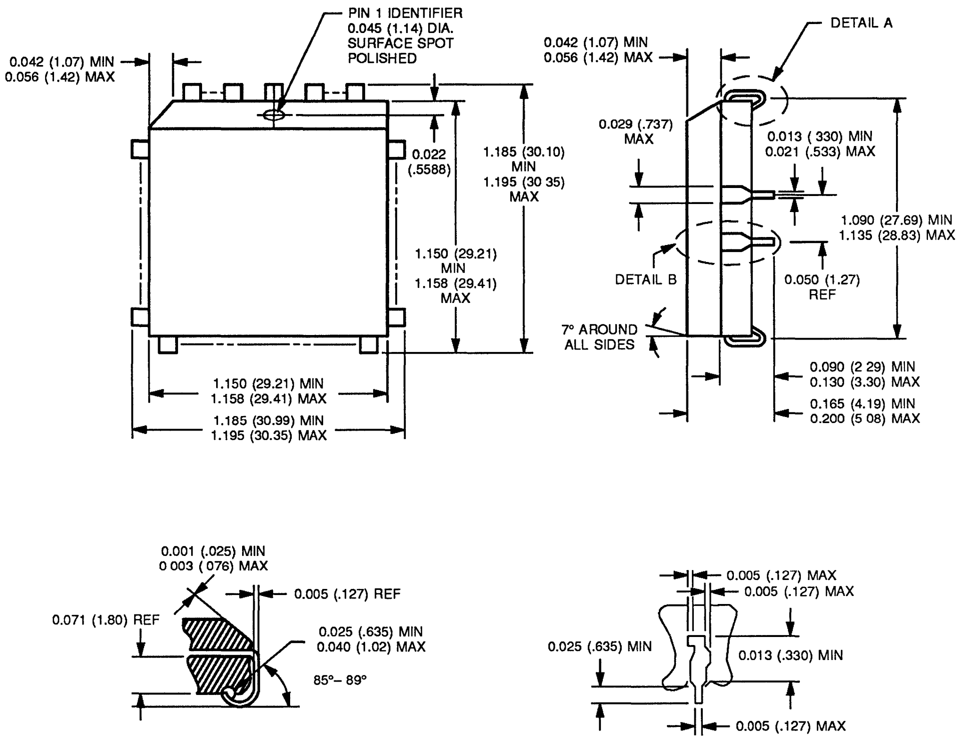
A. Reset Timing



B. Interrupt Timing

Figure 7-3. Reset and Interrupt Timing

7.5 DEVICE PACKAGE OUTLINE



NOTES:
ALL MEET JEDEC STANDARDS.
PIN 1 INDEX MARK MAY BE A DIMPLE OR NUMBERIC LOCATED IN ZONES INDICATED.
DIMENSIONS ARE IN INCHES AND (MILLIMETERS).

Figure 7-4. 84-Pin PLCC Device Outline

Appendix A
Instruction Set Encoding

APPENDIX A. INSTRUCTION SET ENCODING

CONTENTS

A. INSTRUCTION SET ENCODING

This appendix defines the hardware-level encoding of the DSP16/DSP16A device instructions.

A.1 FORMATS

Multiply/ALU Instructions

Format 1: Multiply/ALU Read/Write Group.

Field	T					D	S	F1				X	Y			
Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Format 1a: Multiply/ALU Read/Write Group.

Field	T					$\overline{a}$T	S	F1				X	Y			
Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Format 2: Multiply/ALU Read/Write Group.

Field	T					D	S	F1				X	Z			
Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Format 2a: Multiply/ALU Read/Write Group.

Field	T					$\overline{a}$T	S	F1				X	Z			
Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Special Function Instructions

Format 3: Special Functions.

Field	T					D	S	F2				CON				
Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Control Instructions

Format 4: Branch Direct Group.

Field	T1				JA											
Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Format 5: Branch Indirect Group.

Field	T					B										
Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Format 6: Conditional Branch Qualifier/Software Interrupt (icall).
Note that a branch instruction immediately follows except
for a software interrupt (icall).

Field	T					SI						C				
Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Data Move Instructions

Format 7: Data Move Group.

Field	T					D	R						Y/Z			
Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Format 7a: Data Move Group.

Field	T					$\overline{a}$T	R						Y			
Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Format 8: Data Move (Immediate Operand – 2 words).

Field	T					D	R						Y			
	Immediate Operand (N)															
Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Format 9: Short Immediate Group.

Field	T					I		Short Immediate Operand (M)								
Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Cache Instructions

Format 10: Do – Redo.

Field	T					NI				K						
Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

A.2 REPLACEMENT TABLES FOR FORMAT FIELDS

Field Descriptions

T Field. Specifies the type of instruction.

T	Operation*		Format
0000x	goto JA		4
00010	Short imm j, k, rb, re		9
00011	Short imm r0, r1, r2, r3		9
00100	Y = a1[l]	F1	1
00101	Z : aT[l]	F1	2a
00110	Y	F1	1
00111	aT[l] = Y	F1	1a
01000	aT = R		7a
01001	R = a0		7
01010	R = N		8
01011	R = a1		7
01100	Y = R		7
01101	Z : R		7
01110	do, redo		10
01111	R = Y		7
1000x	call JA		4
10010	ifc CON	F2	3
10011	if CON	F2	3
10100	Y = y[l]	F1	1
10101	Z : y[l]	F1	2
10110	x = Y	F1	1
10111	y[l] = Y	F1	1
11000	Branch indirect		5
11001	y = a0 x = X	F1	1
11010	Cond. branch qualifier		6
11011	y = a1 x = X	F1	1
11100	Y = a0[l]	F1	1
11101	Z : y x = X	F1	2
11110	Reserved		—
11111	y = Y x = X	F1	1

* imm = immediate

D Field. Specifies a destination accumulator.

D	Register
0	Accumulator 0
1	Accumulator 1

$\overline{a}$T Field. Specifies transfer accumulator.

$\overline{a}$T	Register
0	Accumulator 1
1	Accumulator 0

S Field. Specifies a source accumulator.

S	Register
0	Accumulator 0
1	Accumulator 1

X Field. Specifies the addressing of ROM data in two-operand multiply/ALU instructions. Specifies the high or low half of an accumulator or the y register in one-operand multiply/ALU instructions.

X	Operation
Two-Operand Multiply/ALU	
0	*pt++
1	*pt++i
One-Operand Multiply/ALU	
0	aTl, yl
1	aTh, yh

F1 Field. Specifies the multiply/ALU function.

F1	Operation	
0000	aD = p	p = x*y
0001	aD = aS + p	p = x*y
0010		p = x*y
0011	aD = aS – p	p = x*y
0100	aD = p	
0101	aD = aS + p	
0110	NOP	
0111	aD = aS – p	
1000	aD = aS I y	
1001	aD = aS ^ y	
1010	aS & y	
1011	aS – y	
1100	aD = y	
1101	aD = aS + y	
1110	aD = aS & y	
1111	aD = aS – y	

Y Field. Specifies the form of register indirect.

Y	Operation
0000	*r0
0001	*r0++
0010	*r0--
0011	*r0++j
0100	*r1
0101	*r1++
0110	*r1--
0111	*r1++j
1000	*r2
1001	*r2++
1010	*r2--
1011	*r2++j
1100	*r3
1101	*r3++
1110	*r3--
1111	*r3++j

Z Field. Specifies the form of register indirect compound addressing with postmodification.

Z	Operation
0000	*r0zp
0001	*r0pz
0010	*r0m2
0011	*r0jk
0100	*r1zp
0101	*r1pz
0110	*r1m2
0111	*r1jk
1000	*r2zp
1001	*r2pz
1010	*r2m2
1011	*r2jk
1100	*r3zp
1101	*r3pz
1110	*r3m2
1111	*r3jk

I Field. Specifies a register for short immediate data move instructions.

I	Register
00	r0/j
01	r1/k
10	r2/rb
11	r3/re

SI Field. Specifies when the conditional branch qualifier instructions should be interpreted as a software interrupt instruction.

SI	Operation
0	Not a software interrupt
1	Software interrupt

F2 Field. Specifies the special function to be performed.

F2	Operation
0000	aD = aS >> 1
0001	aD = aS << 1
0010	aD = aS >> 4
0011	aD = aS << 4
0100	aD = aS >> 8
0101	aD = aS << 8
0110	aD = aS >> 16
0111	aD = aS << 16
1000	aD = p
1001	aDh = aSh + 1
1010	Reserved
1011	aD = rnd(aS)
1100	aD = y
1101	aD = aS + 1
1110	aD = aS
1111	aD = –aS

B Field. Specifies the type of branch instruction (except software interrupt).

B	Operation
000	return
001	ireturn
010	goto pt
011	call pt
1xx	Reserved

R Field. Specifies the register for data move instructions.

R	Register
000000	r0
000001	r1
000010	r2
000011	r3
000100	j
000101	k
000110	rb
000111	re
001000	pt
001001	pr
001010	pi
001011	i
010000	x
010001	y
010010	yl
010011	auc
010100	psw
010101	c0
010110	c1
010111	c2
011000	sioc
011001	srta
011010	sdx
011011	tdms
011100	pioc
011101	pdx0
011110	pdx1
Other codes	Reserved

C Field. Specifies the condition for special functions and conditional control instructions.

CON	Condition
00000	mi
00001	pl
00010	eq
00011	ne
00100	lvs
00101	lvc
00110	mvs
00111	mvc
01000	heads
01001	tails
01010	c0ge
01011	c0lt
01100	c1ge
01101	c1lt
01110	true
01111	false
10000	gt
10001	le
Other codes	Reserved

NI Field. Number of instructions to be loaded into the cache. Zero implies redo operation.

K Field. Number of times the NI instruction in cache are to be executed.

JA Field. 12-bit jump address.

Appendix B
Programming Examples

APPENDIX B. PROGRAMMING EXAMPLES

CONTENTS

B. PROGRAMMING EXAMPLES

This appendix contains DSP16/DSP16A source-file listings for four complete sample programs. These programs are intended to demonstrate the use of various programming techniques and may be easily modified for use in various applications. Comments within the source files provide all the information necessary to understand the program's function and to make minor modifications.

The programs included are:

- FIR filter
- IIR filter
- 4×4 matrix multiplication
- Find the maximum element in a vector

B.1 FIR FILTER

```
/*                                                                  */
/*      DSP16/DSP16A FIR filter design example:                     */
/*                                                                  */
/*              The following code represents a 66th                */
/*      order FIR filter.  Linear inputs are provided               */
/*      via the serial input.  Linear outputs are sent              */
/*      to the serial output.                                       */
/*                                                                  */
/*              The filter was designed using a Hamming             */
/*      window on an ideal lowpass filter with the following        */
/*      parameters.                                                 */
/*                      f(cutoff) =     1000 Hz                     */
/*                      f(stop) =       1500 Hz                     */
/*                      f(sample) =     10000 Hz                    */
/*                                                                  */
/*      The stopband attenuation is constrained to be               */
/*      above 50 dB.                                                */
/*                                                                  */

.ram
X66:    65*int
X1:     int
ibuf:   int
.endram

/*                                                                  */
/*                      66th order FIR filter                       */
/*                                                                  */
/*              This FIR example uses a modulo addressed            */
/*      delay line.  Outputs are calculated from X(n-66)            */
/*      toward X(n).  The inner loop of the filter uses             */
/*      67 multiplies and requires 89 cycles (6.7 µs at 75 ns),     */
/*      which easily meets the 10000 Hz requirement (100 µs).       */
/*                                                                  */
/*              a0:     scratch calculations                        */
/*              r0:     input buffer location                       */
/*              r1:     delay line pointer                          */
/*              pt:     coefficient pointer                         */
/*                                                                  */
/*      Also modifies rb, re, i, x, y, p, sdx                       */
/*                                                                  */
```

```
fir66:          auc=0x02          /* Use fractional notation        */
                pt=H66            /* Initialize pointers            */
                r0=ibuf
                i=-66
                rb=X66
                re=X1
                r1=X66
loop:           y=0x0010          /* Main loop                      */

wait:           a0=pioc           /* Wait for valid input           */
                a0&y
        if eq   goto wait
                *r0=sdx           /* Input sample                   */

                /*      Perform Convolution     */

                a0=p                      y=*r1++          x=*pt++
        do 65 {
                a0=a0-p           p=x*y   y=*r1++          x=*pt++
        }
                a0=a0-p           p=x*y   y=*r0            x=*pt++i
                a0=a0-p           p=x*y   *r1++=y
                a0=a0-p
                sdx=a0
endl:   goto    loop

/*                                                                  */
/*              Coefficients from h(n-66) to h(n)                   */
/*                                                                  */

H66:    int     -0.000545
        int     -0.000000
        int     0.000641
        int     0.001046
        int     0.000876
        int     -0.000000
        int     -0.001275
        int     -0.002184
        int     -0.001864
        int     -0.000000
        int     0.002673
        int     0.004485
        int     0.003739
        int     -0.000000
        int     -0.005112
        int     -0.008391
        int     -0.006859
        int     -0.000000
        int     0.009085
        int     0.014743
        int     0.011954
        int     -0.000000
        int     -0.015755
        int     -0.025682
        int     -0.021038
        int     -0.000000
        int     0.028989
        int     0.049177
        int     0.042713
        int     -0.000000
        int     -0.073628
        int     -0.157832
        int     -0.224610
        int     -0.250000
        int     -0.224610
        int     -0.157832
        int     -0.073628
        int     -0.000000
        int     0.042713
        int     0.049177
        int     0.028989
        int     -0.000000
        int     -0.021038
        int     -0.025682
        int     -0.015755
        int     -0.000000
        int     0.011954
        int     0.014743
        int     0.009085
        int     -0.000000
        int     -0.006859
        int     -0.008391
        int     -0.005112
        int     -0.000000
        int     0.003739
        int     0.004485
        int     0.002673
        int     -0.000000
        int     -0.001864
        int     -0.002184
        int     -0.001275
        int     -0.000000
        int     0.000876
        int     0.001046
        int     0.000641
        int     -0.000000
        int     -0.000545
```

B.2 IIR FILTER

```
/*                                                                    */
/*      DSP16/DSP16A IIR filter design example:                       */
/*                                                                    */
/*              The following code represents a 5th                   */
/*      order IIR filter. It is implemented as 3 five                 */
/*      multiply second-order sections. (Thus the code is             */
/*      actually the same as a 6th order filter). Linear              */
/*      inputs are provided via the serial input and                  */
/*      prescaled with no loss of precision. Linear outputs           */
/*      are sent to the serial output.                                */
/*                                                                    */
/*              The filter was designed by performing a               */
/*      bilinear transformation on a classical elliptic filter        */
/*      designed with the following parameters.                       */
/*                                                                    */
/*                      f(cutoff) =     1000 Hz                       */
/*                      f(stop) =       1500 Hz                       */
/*                      f(sample) =     10000 Hz                      */
/*                                                                    */
/*      The stopband attenuation is constrained to be                 */
/*      above 50 dB.                                                  */
/*                                                                    */
.ram
statev:         6*int           /* Six state variables                */
.endram

/*              The IIR coding example uses the following             */
/*      DSP16/DSP16A resources:                                       */
/*                                                                    */
/*              a0:     input/output for each section                 */
/*              r1:     state variable pointer                        */
/*              pt:     pointer to filter coefficients                */
/*              j, k:   for compound addressing mode                  */
/*                                                                    */
/*      This coding example demonstrates the use of                   */
/*      compound addressing for maintaining the state                 */
/*      variable delay line.  The form is direct form II.             */
```

```
iir5:          auc=0x02        /* Use fractional notation      */
               j=-2            /* Initialize registers         */
               k=3

loop:          pt=coef         /* Main program loop             */
               r1=statev
               y=0x0010

wait:          a0=pioc         /* Wait for valid input         */
               a0&y
       if eq   goto wait
               a0=sdx
               a0=a0>>1        /* Prescale input value         */
               a0=a0>>1


               /* Perform three second-order sections          */

                                      y=*r1++     x=*pt++     /* a11 */
                          p=x*y       y=*r1--     x=*pt++     /* a21 */
       do 3 {
               a0=a0-p    p=x*y       y=*r1++     x=*pt++     /*  b1(n)   */
               a0=a0-p    p=x*y       *r1zp:y     x=*pt++     /*  b2(n)   */
               a0=p       p=x*y       y=a0        x=*pt++     /*  b0(n)   */
               a0=a0+p    p=x*y       *r1jk:y     x=*pt++     /* a1(n+1) */
               a0=a0+p    p=x*y       y=*r1--     x=*pt++     /* a2(n+1) */
       }
               sdx=a0
endl:  goto    loop

coef:  int     -0.759982              /* a11 */
       int     0.0                    /* a21 */
       int     0.060045               /* b11 */
       int     0.0                    /* b21 */
       int     0.060045               /* b01 */

       int     -1.508821              /* a12 */
       int     0.70049                /* a22 */
       int     -0.108068              /* b12 */
       int     0.245454               /* b22 */
       int     0.245952               /* b02 */

       int     -1.530470              /* a13 */
       int     0.905134               /* a23 */
       int     -1.976028              /* b13 */
       int     1.758388               /* b23 */
       int     1.759004               /* b03 */
```

B.3 MATRIX MULTIPLICATION

```
/*                                                              */
/*      DSP16/DSP16A 4x4 matrix multiply programming example:   */
/*                                                              */
/*      The following code implements a 4x4 matrix multiply.    */
/*      In this example matrices A and B are input through      */
/*      the parallel i/o prior to being multiplied. The result  */
/*      is stored in matrix C. Matrices are input row by row    */
/*      and stored in row order. The resulting C matrix is      */
/*      stored in the same fashion.                             */
/*                                                              */
/*              The matrix multiply routine demonstrates the    */
/*      use of "fast sets" to perform pointer arithmetic        */
/*      both inside and outside of the cache.                   */

/*      Ram variables   */
.ram
A:      16*int
B:      16*int
C:      16*int
.endram


/*              Matrix Multiply routine:                */
/*                                                      */
/*                 r1: points to A                      */
/*                 r0: points to B                      */
/*                 r2: points to C                      */
/*                                                      */
/*              where C = A * B                         */
/*                                                      */
/*      a0, c1, rb, re, j, x, y registers are also      */
/*      used.                                           */
```

```
Mmult:         c1=-3          /* Initialize registers */
               auc=0x00       /* Integer mode         */

/*             Read in Matrices                       */

               r0=A
               a0=pdx0
       do 32 {
               a0=pdx0
               *r0++=a0
             }

               rb=A           /* Reinitialize regs    */
               re=A+3
               r2=C
               r1=A

/*             Main program loop                      */

loop:          r0=B
               j=4
                                      y=*r0++j
                                      x=*r1++
       do 4 {
                           p=x*y      y=*r0++j
               a0=p                   x=*r1++
                           p=x*y      y=*r0++j
               a0=a0+p                x=*r1++
               j=-11
                           p=x*y      y=*r0++j
               a0=a0+p                x=*r1++
               j=4
                           p=x*y      y=*r0++j
               a0=a0+p                x=*r1++
                                      *r2++=a0l
             }
               j=3
                                      *r1++j
               a0=r1
               rb=a0
                                      *r1++j
               a0=r1
               re=a0
                                      *r1++
       if c1lt goto loop
stop:  goto stop
```

B.4 FIND MAXIMUM VECTOR ELEMENT

```
/*              The following segment of code will find the     */
/*      component of an input vector with maximum magnitude.    */
/*      Both the magnitude and index (offset) of the component  */
/*      are retained upon completion.                           */
/*              This coding example demonstrates the use of     */
/*      conditional ALU monadic instructions and event counting */
/*      using the c2 register.                                  */
/*                                                              */

.ram
vector:         32*int
.endram

/*              Read in the vector to be searched via PIO.      */
/*      (It is assumed that the vector will be reused           */
/*      after the determination of its maximum component.       */
/*      Otherwise, a more efficient loop might be written       */
/*      which determines the max component as the values        */
/*      are read.)                                              */

setup:          auc=0x00
                r1=vector
                a0=pdx1
        do 32 {
                a0=pdx1
                *r1++=a0
        }

/*                      Begin search.                           */
/*      Upon completion, the magnitude of the component         */
/*      with maximum value is stored in a0, and the index       */
/*      of this component (its offset from the base address     */
/*      'vector') is stored in c2.                              */
/*              As written, if more than one component of the   */
/*      vector share the maximum value, then the first of these */
/*      two is retained. The routine can easily be modified to  */
/*      retain the last of these components by replacing the    */
/*      test condition with 'le'.                               */

findmaxM:       r1=vector
                c1=0
                c2=0
                                        y=*r1
                                        x=*r1++
                            p=x*y       y=*r1
                a0=p                    x=*r1++
        do 31 {
                            p=x*y       y=*r1
                a1=a0-p                 x=*r1++
                ifc mi  a0=p
              }
stop:   goto stop
```

Appendix C
DSP16/DSP16A Instruction Set Summary

APPENDIX C. DSP16/DSP16A INSTRUCTION SET SUMMARY

CONTENTS

C. DSP16/DSP16A INSTRUCTION SET SUMMARY

Tables 3-2 through 3-12 are repeated below for the convenience of the user, to provide quick access to the DSP16/DSP16A instruction set. See Section 3.9 for a more detailed desciption of the instruction set.

Table C-1. Conditional Mnemonics

Test	Meaning	Test	Meaning
pl	Result is nonnegative (sign bit is bit 35).	mi	Result is negative.
eq	Result is equal to zero.	ne	Result is not equal to zero.
gt	Result is greater than zero.	le	Result is less than or equal to zero.
lvs	Logical overflow set (36-bit overflow).	lvc	Logical overflow clear.
mvs	Mathematical overflow set (32-bit overflow).	mvc	Mathematical overflow clear.
c0ge	Counter 0 greater than or equal to zero.	c0lt	Counter 0 less than zero.
c1ge	Counter 1 greater than or equal to zero.	c1lt	Counter 1 less than zero.
heads	Pseudorandom sequence bit set.	tails	Pseudorandom sequence bit clear.
true	The condition is always satisfied in an **if** instruction.	false	The condition is never satisfied in an **if** instruction.

Note: Testing the state of c0 or c1 automatically increments the counter by 1.

Table C-2. Instruction Group Characteristics

Instruction Group	Flags Affected	Execute From Within Cache	Interruptible
Multiply/ALU	DAU	Yes	Yes
Special Function	DAU	Yes	Yes
Control	None	No	No
Data Move	I/O status only	Yes*	Yes
Cache	None	No	No

* Two-word data immediate instructions may not be executed from within the cache.

All multiply/ALU instructions require one word of memory.

Table C-3. Multiply/ALU Instructions

Function Statements		Transfer Statements	Transfer Cycles Out/In Cache
	p=x*y	y=Y x=X	2/1
aD=p	p=x*y	y=aT x=X	2/1
aD=aS+p	p=x*y	y[l]=Y	1/1
aD=aS-p	p=x*y	aT[l]=Y	1/1
aD=p		x=Y	1/1
aD=aS+p		Y	1/1
aD=aS-p		Y=y[l]	2/2
aD=y		Y=aT[l]	2/2
aD=aS+y		Z: y x=X	2/2
aD=aS-y		Z: y[l]	2/2
aD=aS&y		Z: aT[l]	2/2
aD=aS|y			
aD=aS^y			
aS-y			
aS&y			

Table C-4. Replacement Table for Multiply/ALU Instructions

Replace	Value	Meaning
aD, aS, aT	a0, a1	One of two DAU accumulators.
X	*pt++,*pt++i	ROM location pointed to by pt. pt is postmodified by +1 and i, respectively.
Y	*rM, *rM++, *rM−−, *rM++j	RAM location pointed to by rM. (M= 0, 1, 2, 3) rM is postmodified by 0,+1,−1, and j, respectively.
Z	*rMzp, *rMpz, *rMm2, *rMjk	Read/write compound addressing. M=0, 1, 2, 3. rM is used twice: first, postmodified by 0, +1, −1, and j, respectively; and second, postmodified by +1, 0, +2, and k respectively.

All special function instructions (conditional and unconditional) require 1 word of program memory and execute in 1 instruction cycle.

Table C-5. Special Function Instructions

Instruction	Description

aD=aS>>1
aD=aS>>4
aD=aS>>8
aD=aS>>16
	Arithmetic right shift (sign preserved) of 36-bit accumulators.

aD=aS
aD=–aS
	—

aD=rnd(aS)	Round upper 20 bits of accumulator.
aDh=aSh+1	Increment high half of accumulator (lower half cleared).
aD=aS+1	Increment accumulator.

aD=y
aD=p
	—

aD=aS<<1
aD=aS<<4
aD=aS<<8
aD=aS<<16
	Logical left shift (sign-extended from bit 31) of the least significant 32 bits of the 36-bit accumulators.

The above special function instructions can be conditionally executed:

if CON *instruction*

and with an event counter:

ifc CON *instruction*

which means:

```
if CON is true then
    c1 = c1 + 1
    instruction
    c2 = c1
else
    c1 = c1 + 1
```

Table C-6. Replacement Table for Special Function Instructions

Replace	Value	Meaning
aD,aS	a0,a1	One of two DAU accumulators.
CON	mi, pl, eq, ne, gt, le, lvs, mvs, mvc, c0ge, c0lt, c1ge, c1lt, heads, tails, true, false	See Table C-1 for definitions of processor flags.

All unconditional control instructions (except icall) execute in 2 instruction cycles and require 1 word of program memory; conditional control instructions execute in 3 instruction cycles and require 2 words of program memory. **icall** requires 1 word of program memory and executes in 3 instruction cycles.

Table C-7. Control Instructions

goto *JA*	icall
goto pt	return (goto pr)
call *JA*	ireturn (goto pi)
call pt	

The control instructions, with the exception of ireturn and icall can be conditionally executed as follows:

if CON *instruction*

Table C-8. Replacement Table for Control Function Instructions

Replace	Value	Meaning
CON	mi, pl, eq, ne, gt, le, lvs, mvs, mvc, c0ge, c0lt, c1ge, c1lt, heads, tails, true, false	See Table 3-12 for definitions of processor flags.
JA	12-bit value	Least significant 12 bits of an absolute address within the same 4 Kword memory section.

Data move instructions execute in 2 instruction cycles. Immediate data move instructions require two words of program memory; all other data move instructions require only 1 word. The only exception is a special case immediate load (short immediate) instruction. If a YAAU register is loaded with a 9-bit, or smaller, value, the instruction requires only 1 word of memory and executes in 1 instruction cycle.

Table C-9. Data Move Instructions

R = N	R = M
R = Y	Y = R
aT = R	R = aS
Z : R	

Table C-10. Replacement Table for Data Move Instructions

Replace	Value	Meaning
R	x	DAU register – signed, 16 bits.
	y	DAU register – signed, 16 bits.[1]
	yl	DAU register – unsigned, 16 bits.
	auc	DAU control register – unsigned, 7 bits.
	c0	DAU counter 0 – signed, 8 bits.
	c1	DAU counter 1 – signed, 8 bits.
	c2	DAU counter 2 – signed, 8 bits.
	r0	YAAU pointer reg. – unsigned, 9 bits (16 bits in DSP16A).
	r1	YAAU pointer reg. – unsigned, 9 bits (16 bits in DSP16A).
	r2	YAAU pointer reg. – unsigned, 9 bits (16 bits in DSP16A).
	r3	YAAU pointer reg. – unsigned, 9 bits (16 bits in DSP16A).
	rb	YAAU modulo addr. reg. – unsigned, 9 bits (16 bits in DSP16A).
	re	YAAU modulo addr. reg. – unsigned, 9 bits (16 bits in DSP16A).
	j	YAAU incr. register – signed, 9 bits (16 bits in DSP16A).
	k	YAAU incr. register – signed, 9 bits (16 bits in DSP16A).
	pt	XAAU pointer register – unsigned, 16 bits.
	pr	XAAU program return register – unsigned, 16 bits.
	pi	XAAU program interrupt register – unsigned, 16 bits.[2]
	i	XAAU incrmient register — signed, 12 bits.
	psw	Processor status word.
	sioc	Serial I/O control register.[3]
	sdx	Serial I/O data register.
	tdms	Serial I/O tdms control register.[3]
	srta	Serial receive/transmit address.[3]
	pioc	Parallel I/O control register.
	pdx0	Parallel I/O data register with PSEL = 0 (pin 72).
	pdx1	Parallel I/O data register with PSEL = 1 (pin 72).
aD, aS	a0, a1	High half of accumulator.[1]
Y	*rM,*rM++, *rM− −,*rM++j	Same as in multiply/ALU instructions.
Z	*rMzp,*rMpz, *rMm2,*rMjk	Same as in multiply/ALU instructions.
N	16-bit value	Immediate data.
M	9-bit value	Immediate data for YAAU registers.

Notes:

When reading signed registers less than 16 bits wide, their contents are sign-extended to 16 bits. When reading unsigned registers less than 16 bits wide, their contents are zero-extended to 16 bits. When short immediate addressing is used to write to YAAU registers in the DSP16A, unsigned registers are zero-extended from 9 to 16 bits. Signed registers (j,k) are sign-extended from 9 to 16 bits.

[1]Data moves to y, a0, or a1 load the high half (bits 31—16) of the register. If clearing of the destination is enabled (according to the CLR field of the auc register), the low half of the destination register is cleared (0) when the high half is loaded.

[2]The pi register acts as a "shadow" of the pc register. Each time the pc changes, its value is also loaded into pi. "Shadowing" is disabled when executing an interrupt service routine, therefore, pi contains the contents of pc prior to the interrupt. Writes to pi do not alter its contents, except during interrupt service routines.

[3]sioc, tdms, and srta registers are not readable.

The **do** and **redo** instructions require 1 word of program memory. The **do** instruction executes in 1 instruction cycle, and the **redo** instruction executes in 2 instruction cycles.

Table C-11. Cache Instructions
do *K* **{** *instruction1* *instruction2* *instructionNI* **}**
redo *K*

Table C-12. Replacement Table for Cache Instructions

Replace	**Value**	**Meaning**
K	$2 \leq K \leq 127$	Number of times the instructions are to be executed.
NI	$1 \leq NI \leq 15$	1 to 15 instructions may be included.

When the cache is used to repeat a block of NI instructions, the cycle timings of the instructions are as follows:

1. The "first pass" does not affect cycle timing except for the last instruction in the block of NI instructions. This instruction executes in 2 cycles.
2. During pass 2 through pass K+1, each instruction is executed "in the cache" (see Table C-3).
3. During the last (Kth) pass, the block of instructions executes "inside the cache" except for the last instruction which executes outside the cache.

The instructions remain in the cache memory and may be re-executed using the **redo** command without the need to reload the cache.

Appendix D
Programmable Registers

APPENDIX D. PROGRAMMABLE REGISTERS

CONTENTS

D. PROGRAMMABLE REGISTERS

This reference section shows the six programmable control registers of the DSP16/DSP16A device:

- Processor status word (psw)
- Arithmetic unit control (auc)
- Parallel I/O control (pioc)
- Serial I/O control (sioc)
- Serial receive/transmit address (srta)
- Time-division multiplexed slot (tdms)

All six registers are described in detail in other chapters in this manual. This section is provided only as a quick reference for the programmable registers.

Table D-1. Arithmetic Unit Control (auc) Register

Bit	6	5	4	3	2	1	0
Field	CLR			SAT		ALIGN	

Field	Value	Result/Description
CLR	1xx	Clearing yl is disabled (enabled when 0).
	x1x	Clearing a1l is disabled (enabled when 0).
	xx1	Clearing a0l is disabled (enabled when 0).
SAT	1x	a1 saturation on overflow is disabled (enabled when 0).
	x1	a0 saturation on overflow is disabled (enabled when 0).
ALIGN	00	$p \leftarrow (x \times y)$.
	01	$p \leftarrow (x \times y) \div 4$.
	10	$p \leftarrow (x \times y) \times 4$.
	11	Reserved.

Table D-2. Processor Status Word (psw) Register

Bit	15—12	11	10	9	8—5	4	3—0
Field	DAU Flags	X	X	a1[V]	a1[35–32]	a0[V]	a0[35–32]

Field	Value	Result/Description
DAU Flags	Wxxx	lMI – logical minus when set.
	xWxx	LEQ – logical equal when set.
	xxWx	LLV – logical overflow when set.
	xxxW	LMV – mathematical overflow when set.
a1[V]	W	Accumulator 1 (a1) overflow when set.
a1[35–32]	Wxxx	Accumulator 1 (a1) bit 35.
	xWxx	Accumulator 1 (a1) bit 34.
	xxWx	Accumulator 1 (a1) bit 33.
	xxxW	Accumulator 1 (a1) bit 32.
a0[V]	W	Accumulator 0 (a0) overflow when set.
a0[35–32]	Wxxx	Accumulator 0 (a0) bit 35.
	xWxx	Accumulator 0 (a0) bit 34.
	xxWx	Accumulator 0 (a0) bit 33.
	xxxW	Accumulator 0 (a0) bit 32.

Table D-3. Parallel I/O Control (pioc) Register

Bit	15	14 13	12	11	10	9—5	4—0
Field	IBF	STROBE	PODS	PIDS	S/C	INTERRUPTS	STATUS

Field	Value	Result/Description
IBF	R	IBF interrupt status bit (same as bit 4).
STROBE		Strobe width of **PODS PIDS**
	00	T* T
	01	2T 2T
	10	3T 3T
	11	4T 4T
PODS	0	PODS is an input (passive mode).
	1	PODS is an output (active mode).
PIDS	0	PIDS is an input (passive mode).
	1	PIDS is an output (active mode).
S/C	0	Not S/C mode.
	1	S/C mode.
INTERRUPTS	Wxxxx	IBF interrupt enabled when set.
	xWxxx	OBE interrupt enabled when set.
	xxWxx	PIDS interrupt enabled when set.
	xxxWx	PODS interrupt enabled when set.
	xxxxW	INT interrupt enabled when set.
STATUS	Rxxxx	IBF status bit.
	xRxxx	OBE status bit.
	xxRxx	PIDS status bit.
	xxxRx	PODS status bit.
	xxxxR	INT status bit.

* T = 2xtCKIHCKIH.

Table D-4. Serial I/O Control (sioc) Register

Bit	9	8	7	6	5	4	3	2	1	0
Field	LD	CLK		MSB	OLD	ILD	OCK	ICK	OLEN	ILEN

Field	Value	Result/Description
LD	0	Active ILD/OLD = ICK÷16, Active SYNC = ICK÷128/256†
	1	Active ILD/OLD = OCK÷16,‡ Active SYNC = OCK÷128/256.†,‡
CLK	0 0	Active clock = CKI÷4
	0 1	Active clock = CKI÷12
	1 0	Active clock = CKI÷16
	1 1	Active clock = CKI÷20
MSB	0	LSB first
	1	MSB first
OLD	0	OLD is an input (passive mode).
	1	OLD is an output (active mode).
ILD	0	ILD is an input (passive mode).
	1	ILD is an output (active mode).
OCK	0	OCK is an input (passive mode).
	1	OCK is an output (active mode).
ICK	0	ICK is an input (passive mode).
	1	ICK is an output (active mode).
OLEN	0	16-bit output
	1	8-bit output
ILEN	0	16-bit input
	1	8-bit input

† Either 128 or 256 – see tdms register SYNC field.
‡ Select this mode when using SADD (not necessary if ICK = OCK).

Table D-5. Serial Receive/Transmit Address (srta) Register

Bit	15 — 8	7 — 0
Field	RECEIVE ADDRESS	TRANSMIT ADDRESS

Field	Value	Result/Description
RECEIVE ADDRESS	1xxxxxxx	Receive address 7
	x1xxxxxx	Receive address 6
	xx1xxxxx	Receive address 5
	xxx1xxxx	Receive address 4
	xxxx1xxx	Receive address 3
	xxxxx1xx	Receive address 2
	xxxxxx1x	Receive address 1
	xxxxxxx1	Receive address 0
TRANSMIT ADDRESS	1xxxxxxx	Transmit address 7
	x1xxxxxx	Transmit address 6
	xx1xxxxx	Transmit address 5
	xxx1xxxx	Transmit address 4
	xxxx1xxx	Transmit address 3
	xxxxx1xx	Transmit address 2
	xxxxxx1x	Transmit address 1
	xxxxxxx1	Transmit address 0

Table D-6. Time-Division Multiplex Slot (tdms) Register

Bit	9	8	7 — 1	0
Field	SYNCSP	MODE	TRANSMIT SLOT	SYNC

Field	Value	Result/Description
SYNCSP	0	SYNC = ICK/OCK† ÷ 128 ‡
	1	SYNC = ICK/OCK† ÷ 256
MODE	0	Multiprocessor mode off. DOEN is an input (passive mode).
	1	Multiprocessor mode on. DOEN is an output (active mode).
TRANSMIT SLOT	1xxxxxxx	Transmit slot 7.
	x1xxxxxx	Transmit slot 6.
	xx1xxxxx	Transmit slot 5.
	xxx1xxxx	Transmit slot 4.
	xxxx1xxx	Transmit slot 3.
	xxxxx1xx	Transmit slot 2.
	xxxxxx1x	Transmit slot 1.
SYNC	xxxxxxx1	Transmit slot 0. SYNC is an output (active mode).
	xxxxxxx0	SYNC is an input (passive mode).

† See sioc register, LD field in Table D-4.

‡ Select this mode when in multiprocessor mode.

Glossary

A-law – A European standard for the compression and expansion of the dynamic range of a signal.

Active mode – Certain pins on the DSP16/DSP16A device are programmable as either inputs or outputs. When one of these signals is set as an output, it is in active mode. When one of these signals is set as an input, it is in passive mode.

ADD – Adder.

Addressing modes – The DSP16/DSP16A device supports various modes for addressing, which include immediate, indirect, and compound.

ALU – Arithmetic logic unit.

Arithmetic logic unit (ALU) – On-chip unit that performs microprocessor-like arithmetic operations.

Arithmetic unit control register (auc) – A register that configures some features for the data arithmetic unit.

Assembler – A program that translates symbolically represented character input into a form (binary) that the computer can interpret.

auc – Arithmetic unit control register.

Breakpointing – The ability to have the software simulator or hardware development system halt and/or perform some command at a present location or when some test condition is met.

Byte – An 8-bit quantity that may appear at any address in memory.

Cache – A small, high-speed memory that can be used selectively to store repetitive operations.

Compound Addressing – A memory read/write operation using only one pointer.

CMP – Comparator.

Cyclical addressing – See **Modulo addressing**.

Data arithmetic unit (DAU) – A 16-/32-bit unit that is the main execution unit for signal processing algorithms.

DAU – Data arithmetic unit.

Hardware mode – When using the DSP16 or DSP16A Development System, a mode in which the program has been downloaded into the development system and is being executed by the actual DSP16/DSP16A device. Hardware mode is used for real-time program testing.

I – Increment register in XAAU.

ibuf – Input buffer.

Input buffer (ibuf) – A register in the SIO unit is used to accept input from an external device.

Immediate addressing mode – An addressing mode in which the operand contains the value to be operated on. No address reference is required.

Interrupt – A means by which external devices may request service by the microprocessor.

I/O – Input/output.

isr – Input shift register.

j – Increment register in YAAU.

k – Increment register in YAAU.

Latency – The time required for the completion of a task once initiated.

Modulo addressing – Cyclical addressing between upper and lower bounds.

µ-law – An American standard for compression and expansion of the dynamic range of a signal.

MUX – Multiplexer.

osr – Output shift register.

Overhead – A quantity in addition to the minimum required. When transmitting data, it is the bits other than information bits, e.g., check bits, framing bits, or some other procedure or format bits. When executing a program loop, it is the instructions needed to control the program flow, those not involved in the desired computation.

p – Product register.

Parallel I/O port (pio) – A group of registers used to provide a bidirectional data link to microprocessors and other I/O devices.

Parallel I/O control register (pioc) – A register that allows the specification of the configuration of the PIO data pins, the mode of the parallel I/O data strobe signals, and the width of the parallel I/O data strobe signals in the active mode. The pioc also contains the mask and status fields for interrupts.

Passive mode – Certain pins on the DSP16/DSP16A device are programmable as either inputs or outputs. When one of these signals is set as an input, it is in passive mode. When one of these signals is set as an output, it is in active mode.

pc – Program counter.

pdx(in) – Parallel I/O input register.

pdx(out) – Parallel I/O output register.

pi – Program interrupt register.

pio – Parallel I/O port.

pioc – Parallel I/O control register.

Pipelining – Overlapping the execution of instructions to increase the DSP's performance.

pr – Program return register.

Processor status word register (psw) – A register that contains the status information for the data arithmetic unit.

Postmodification – The addition of an increment/decrement value to a memory pointer after each use.

psw – Processor status word.

pt – ROM table pointer.

RAM – Random access memory.

rb – Modulo addressing register containing the beginning value of the modulo.

re – Modulo addressing register containing the end value of the modulo. re contains zero if modulo addressing is not active.

ROM – Read only memory.

sdx(in) – Serial I/O input register.

sdx(out) – Serial I/O output register.

Serial I/O (SIO) – A group of registers that allows interfacing with other devices with few, if any, external chips. It converts serial input data into parallel data and parallel data into serial output data.

Serial I/O control register (sioc) – A register that allows specification of the length of serial input and output data words, the mode of serial bit clocks and serial load signals, the bit ordering of I/O, the active serial I/O bit clock rate, and active load generated from either ICK or OCK.

Simulator – A highly specific program that allows the simulation in software of the logical functions of the DSP16/DSP16A device.

SIO – Serial I/O unit.

sioc – Serial I/O control register.

srta – Serial receive/transmit address register.

Stack – An area of reserved memory used for storing the program counter and contents of registers during a program interrupt.

SHIFT – Shifter.

tdms – Serial I/O time-division multiplexed slot register.

Throughput – A means of relating the speed with which problems, programs, or segments are performed.

Time-division multiplexed (TDM) – A procedure for transmitting two or more signals over a common channel through the use of successive time intervals for different devices.

x – Multiplier input register.

XAAU – ROM address arithmetic unit.

YAAU – RAM address arithmetic unit.

y or yh – y(high) DAU register.

yl – y(low) DAU register.

2's complement – A method used in some systems to represent positive and negative integers. Positive integers are identical to standard binary numbers; however, negative integers are the 1's complement of a standard binary number plus one.

3-state – To place an output in a high-impedance state.

Index

For additional information, contact your AT&T Account Manager, or call:

☐ AT&T Microelectronics
Dept. 50AL330240
555 Union Boulevard
Allentown, PA 18103
1-800-372-2447

In Canada, call:
1-800-553-2448

☐ AT&T Microelectronics
AT&T Deutschland GmbH
Bahnhofstr. 27A
D-8043 Unterfoehring
West Germany
Tel. 089/950 86-0
Telefax 089/950 86-111

☐ AT&T Microelectronics Asia/Pacific
14 Science Park Drive
#03-02A/04 The Maxwell
Singapore 0511
Tel. (65) 778-8833
FAX (65) 777-7495
Telex RS 42898 ATTM

☐ AT&T Microelectronics
AT&T Japan Ltd.
31-11, Yoyogi 1-chome
Shibuya-ku, Tokyo 151
Japan
Tel. (03) 5371-2700
FAX (03) 5371-3556

☐ AT&T Microelectronica España
Albacete 5
28027 Madrid
Spain
Tel. (34) 1-404-6012
FAX (34) 1-404-6252

January 1989

MN38-18DMOS

AT&T
The right choice.