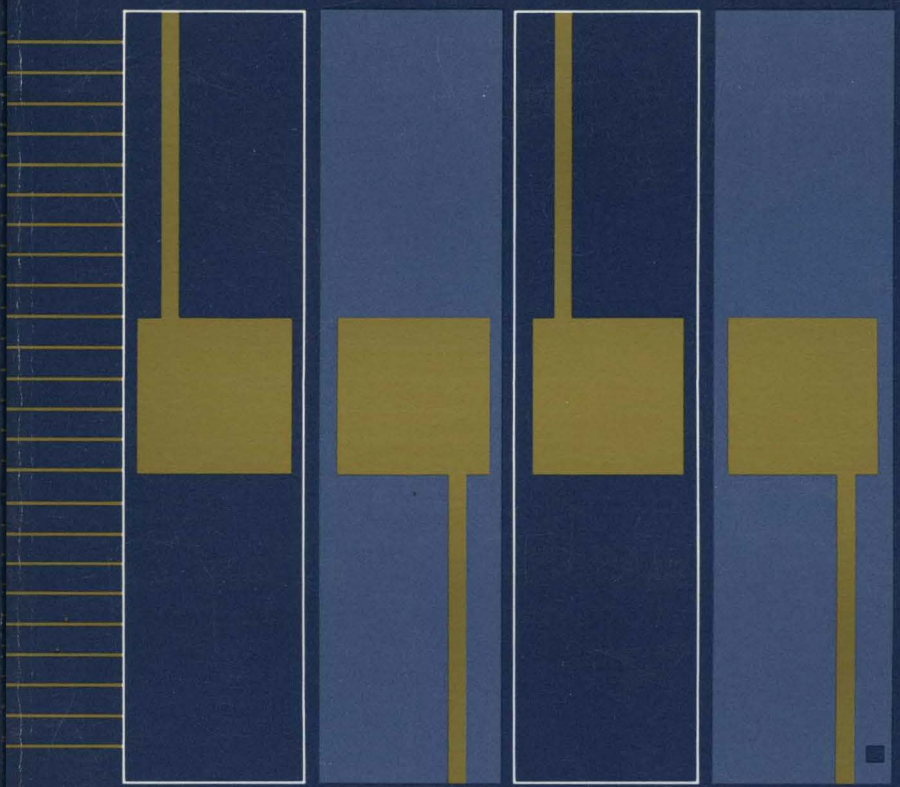
Word–Slice™ User's Manual ■ ADSP-1401 / ADSP-1410

ANALOG DEVICES



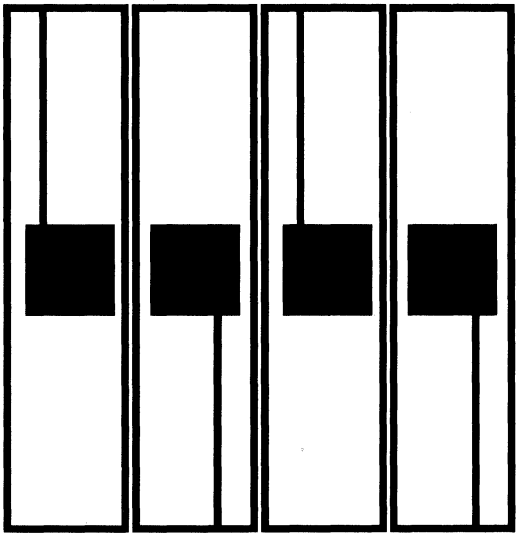
ANALOG DEVICES

Word–Slice™ User's Manual

ADSP-1401 Program Sequencer

ADSP-1410 Address Generator

Word-Slice User's Manual



ADSP-1401 Program Sequencer

ADSP-1410 Address Generator

Word-Slice™ User's Manual

Printed in USA.

Contents

Contents

Contents

Contents

ADSP-1410: CHAPTERS 7 – 11

CHAPTER 7 INTERNAL ARCHITECTURE

Contents

Contents

Contents

LIST OF FIGURES

Contents

LIST OF TABLES

Introduction 1

1.1 SYSTEM OVERVIEW

The ADSP-1401 Program Sequencer and the ADSP-1410 Address Generator form the Word-Slice™ chipset for implementing microcoded designs. A typical Word-Slice system is shown in Figure 1.1. The ADSP-1401 addresses the microcode memory, which provides instructions for all of the system components. Arithmetic processing can be provided by one or more computational devices, such as Analog Devices' fixed-point or floating-point ALUs and multipliers. Data transfers between the processing units and the data/coefficient memory are facilitated through addressing generated by one or more ADSP-1410s. Note that both the ADSP-1401 and the ADSP-1410 latch instructions internally, whereas the arithmetic processing units use external instruction latches.

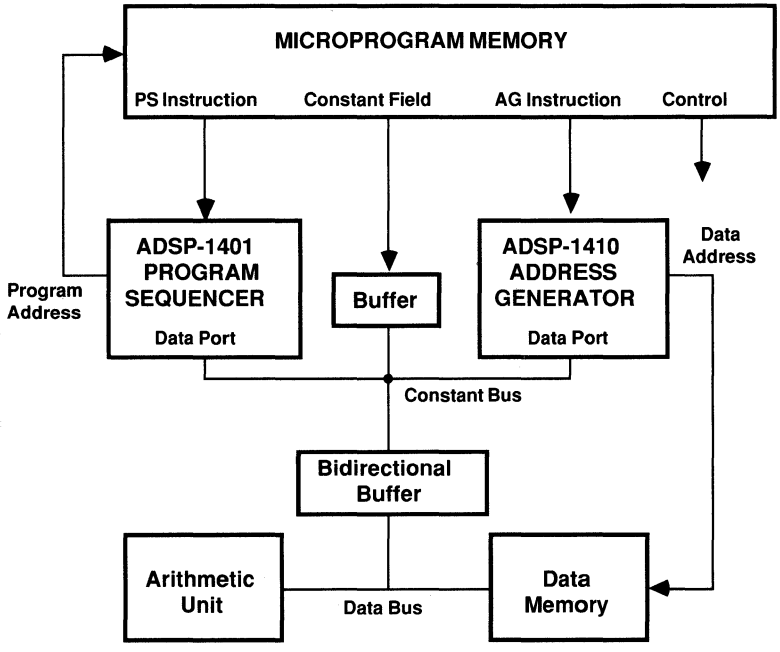


Figure 1.1 Typical Word-Slice System

1 Introduction

1.2 ADSP-1401 OVERVIEW

The ADSP-1401 Program Sequencer provides the addresses needed to sequence a microcoded system through the instructions stored in microcode memory. It can output 16-bit addresses at a high speed because its internal Look-Ahead™ pipeline coordinates the timing of its instruction input and address output, which are both latched on-chip. Because of this timing, the ADSP-1401 never incurs a delay performing a jump; the jump instruction is decoded *before* the next address is output.

ADSP-1401 instructions are only seven bits wide to minimize its instruction space in microcode. At the same time, these instructions are powerful, reducing the programming requirements for complex operations. Besides basic straight-line execution of code, the ADSP-1401 supports conditional and unconditional jumps, including subroutine and interrupt service routine jumps and returns. Four decrementing counters store twos-complement values whose sign bits can be used as conditions for jump execution. An external Flag input provides another jump condition. The ADSP-1401 can read an address or offset at its data port for absolute, relative, or indirect addressing.

On-chip RAM stores return addresses, jump addresses, registers values, and local variables. The RAM capacity is 64 words, large enough to accommodate many levels of subroutine nesting. The RAM consists of three sections: the subroutine stack, the register stack, and the indirect jump address space. The relative sizes of the three sections are user-configurable.

The ADSP-1401 provides complete interrupt processing on-chip. It stores vectors for eight external and two internal interrupts. The dedicated internal interrupts indicate stack limit violations and counter underflow. Interrupts can be individually masked, or disabled and enabled as a group. Interrupts are latched to ensure recognition even if the interrupt source removes the signal. This latching can be disabled (transparent) if not needed or desired. The ADSP-1401 handles an interrupt by outputting the corresponding vector and saving the return address.

The ADSP-1401 can be placed in a Writeable Control Store mode in which it provides sequential addressing to download instructions into microcode from an external source (without executing the downloaded instructions). In this mode, the ADSP-1401 conforms to download timing requirements through a handshake signal.

1.3 ADSP-1410 OVERVIEW

The ADSP-1410 Address Generator produces the data memory addresses needed to implement digital signal processing and array processing algorithms quickly and efficiently. It can simultaneously output an address from one of its 16 address registers and calculate the address of the next data access in a single instruction cycle. The ADSP-1410 can also determine whether an address exceeds a user-defined limit and then reset the address value accordingly. This feature allows it to generate looping address sequences without overhead penalty.

The ADSP-1410 has an internal ALU that can update an existing address register through following operations:

- Increment
- Decrement
- Add offset value
- Subtract offset value
- OR with preloaded value
- AND with preloaded value
- XOR with preloaded value
- Shift one bit left
- Shift one bit right

The ADSP-1410 can also bit-reverse addresses upon output.

A single ADSP-1410 provides 16-bit addresses; you can expand the addressing capability to 30 bits by either cascading two ADSP-1410 devices or executing two cycles for each address. A 16-bit bidirectional data port lets you transfer data to and from other devices. You can also transfer data between internal registers using dedicated instructions.

The internal Look-Ahead™ pipeline latches the instruction input port and address output port in opposite phases of the clock cycle. This timing allows the ADSP-1410 to begin decoding the next instruction while maintaining the address of the current cycle. Predecoding instructions provides the fastest possible address output, for the maximum possible memory access time. If memory access time is not critical, however, you can forego predecoding and reduce the instruction setup time requirement.

The Alternate Instruction Register (AIR) of the ADSP-1410 provides a single-instruction cache that can reduce microcode memory requirements for many applications. For example, if an instruction must be repeated, it would occupy several locations in microcode. You can store it in the AIR instead and enable the AIR the number of times needed. The AIR can also be used to supersede the instruction in microcode conditionally (to exit a loop, for example). Modulo addressing, which is described in Chapter 8, is accomplished using conditional AIR execution.

1 Introduction

1.4 THIRD-PARTY SUPPORT

A number of third-party vendors produce development tools for Word-Slice components. At the time of this publication, these tools include the following:

- METASTEP is a meta-assembler package from STEP Engineering that eases the task of writing microcode programs for Word-Slice components. This software runs on a variety of host computers, including the IBM PC and the VAX. STEP Engineering also sells its own development systems.

- Microtec Research supports Word-Slice components with two meta-assemblers. Their META29R Relocatable Macro Meta-Assembler System consists of a definition program, an assembler, linker, and PROM formatter. META29R is Microtec's version of AMDASM. Their mcASM, a second-generation meta-assembler, can use files created by AMDASM and META29R but provides a high-level approach. Definition files for Word-Slice components are included with both meta-assemblers.

- HILEVEL Technology's HALE (HILEVEL Assembly Language Environment) supports both Word-Slice components and Analog Devices' floating-point components. HILEVEL also has two control store boxes: the DS370 Emulyzer and the DS3700 Emulyzer.

- The Hewlett-Packard (HP 64276) Microprogram Development Subsystem in conjunction with their HP 64320S Logic State/Software Analyzer provides control store emulation. The HP 64276 includes a meta-assembler which is not capable of storing permanent definition files. All of these tools require an HP 64110A development workstation as a host.

- Logic Automation's SmartModels are software behavioral models of various devices, including the Word-Slice components. Several simulation programs generate board-level simulations using these models. Prototyping time is thereby greatly reduced.

For more information, contact the vendors directly at the following addresses:

STEP Engineering
P.O. Box 61166
Sunnyvale, CA 94088
800-538-1750

Microtec Research
P.O. Box 60337
Sunnyvale, CA 94088
408-733-2919

HILEVEL Technology
18902 Bardeen Way
Irvine, CA 92715
800-HILEVEL

Logic Automation
P.O. Box 310
Beaverton, OR 97075
503-690-6900

Hewlett-Packard
(Contact a local sales office)

1.5 MANUAL ORGANIZATION

This manual is organized into two main sections, indicated by two banks of tabs that mark the chapters. The first section (Chapters 2 through 6) is devoted to the ADSP-1401; the second(Chapters 7 through 11) to the ADSP-1410. The chapters contain the following information:

- Chapter 1 describes the general features of the ADSP-1401 and the ADSP-1410 as well as the organization of this manual.
- Chapter 2 describes the internal architecture of the ADSP-1401. The function of each area of the device is explained, as well as the interrelationships between various areas.
- Chapter 3 describes how to implement program jumps, including loops and subroutines. Jump conditioning and addressing options are explained.
- Chapter 4 describes the interrupt processing capabilities of the ADSP-1401. The ADSP-1401 can handle up to eight external interrupts plus two interrupts generated internally.
- Chapter 5 describes the system interface of the ADSP-1401. Connections to and from the device are explained.
- Chapter 6 describes the instruction set of the ADSP-1401. The mnemonic, opcode, and description for each instruction are listed. Short examples are given for each instruction. A summary section provides the mnemonic, opcode, and a short description for each instruction.

ADSP-1401

- Chapter 7 describes the internal architecture of the ADSP-1410. The function of each area of the device is explained, as well as the interrelationships between various areas.
- Chapter 8 describes various addressing operations performed by the ADSP-1410, including how it selects the address to output and how it updates the address to anticipate the next access.
- Chapter 9 describes the precision modes of the ADSP-1410. Precision modes determine the width of the generated addresses.
- Chapter 10 describes the system interface of the ADSP-1410. Connections to and from the device are explained.
- Chapter 11 describes the instruction set of the ADSP-1410. The reference section of this chapter describes the instructions in detail and gives short examples. A summary section provides the mnemonic, opcode, and a short description for each instruction.

ADSP-1410

Internal Architecture 2

2.1 INTRODUCTION

The ADSP-1401 is a 48-pin CMOS device. Pin names and definitions are listed in Table 2.1.

PIN NAME	*DEFINITION*
I_{6-0}	Instruction input, seven bits
Y_{15-0}	Address output, 16 bits
D_{15-0}	Data I/O, 16 bits
$EXIR_{4-1}$	External interrupts, four inputs that are time-multiplexed to generate eight internal signals (IR_{8-1})
FLAG	Condition input
TTR	Three-function control input that is time-multiplexed to generate internal Trap, Tristate, and Reset signals
CLK	Clock input
V_{DD}	+5 Volt supply
GND	Ground

Table 2.1 Pin Definitions

Figure 2.1 shows a block diagram of the ADSP-1401. The device consists of the following major areas, which are described in the sections of this chapter:

- Instruction Port and Instruction Decoder
- Bidirectional Data Port
- Program Counter
- Adder and Width Control
- Internal RAM and Stack Pointers
- Interrupt Logic and Interrupt Vector File
- Event Counters
- Flag
- Trap/Tristate/Reset
- Address Multiplexer and Address Port
- Status Register

2 Internal Architecture

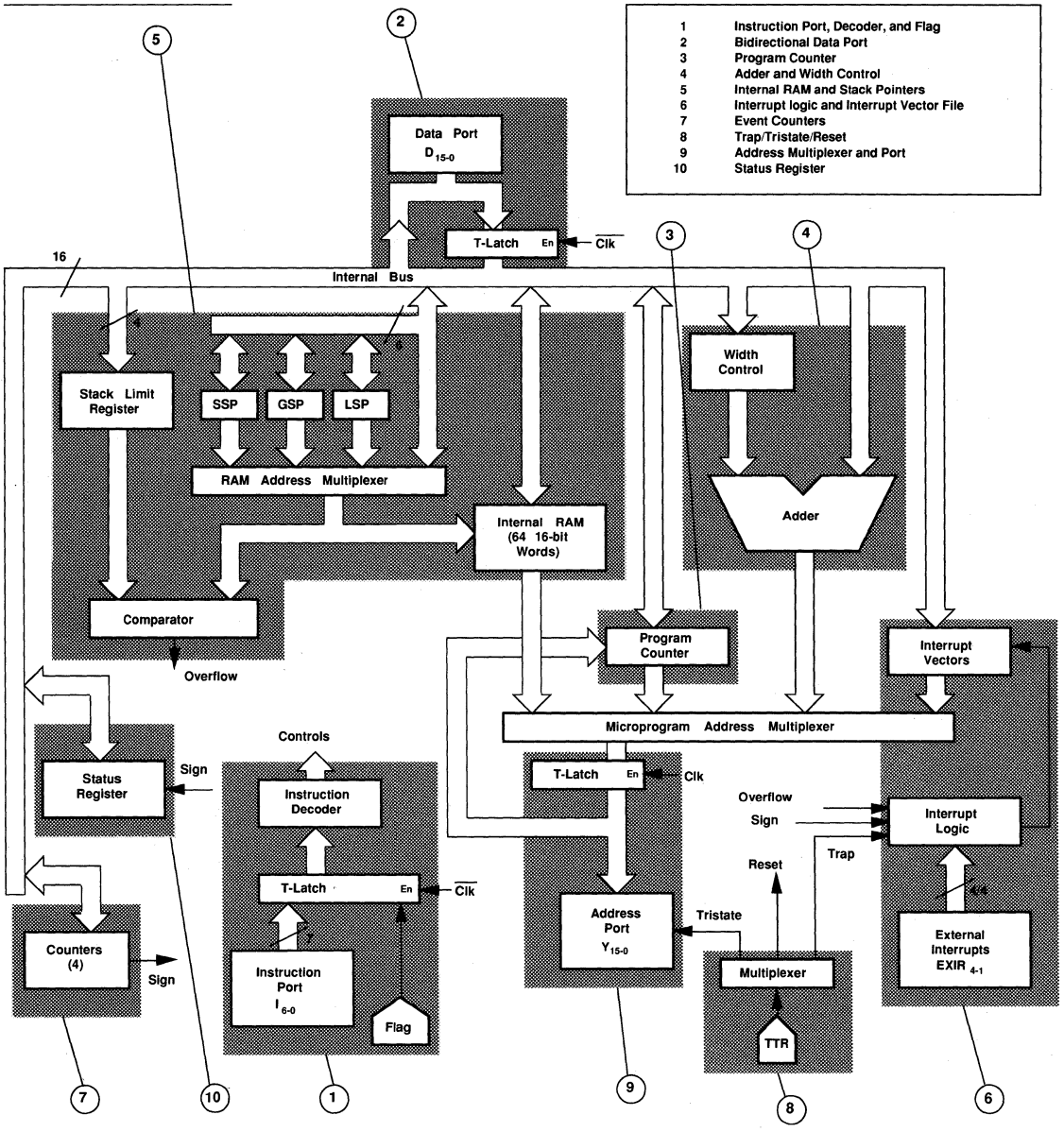


Figure 2.1 ADSP-1401 Block Diagram

Internal Architecture 2

2.2 INSTRUCTION PORT

The ADSP-1401 receives instructions through its instruction port, which consists of seven pins (I_{6-0}). The ADSP-1401 latches an instruction during clock HI. The instruction port is transparent during clock LO to allow predecoding of the next instruction, as shown in Figure 2.2.

You can program the ADSP-1401 to repeat an instruction rather than latch in a new instruction. See *Instruction Hold Control* in Chapter 5 for more information.

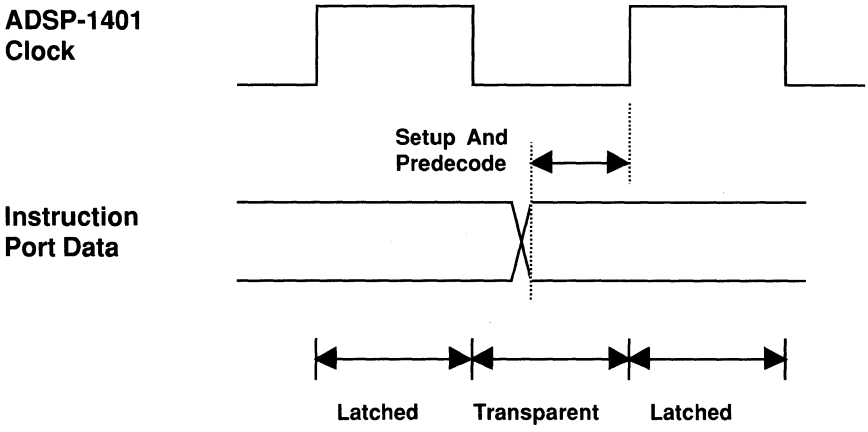


Figure 2.2 Instruction Port Latching

2.3 BIDIRECTIONAL DATA PORT

The bidirectional data port consists of 16 I/O pins (D_{15-0}) through which the ADSP-1401 loads values for direct or indirect addressing and transfers register values. Data input is latched during clock HI. Data output is enabled during clock HI and disabled during clock LO. This timing accommodates a data output followed by a data input in the next cycle, as shown in Figure 2.3. To prevent contention on the data bus, the ADSP-1401 disables data output in time to allow the data input to meet the required setup to the rising clock edge.

2 Internal Architecture

2.4 PROGRAM COUNTER

The program counter provides addressing for sequential program execution. The ADSP-1401 increments its 16-bit program counter at the end of every instruction cycle, unless the instruction specifically inhibits the increment. Jump instructions load the program counter with the jump address, so that sequential execution can continue after the jump.

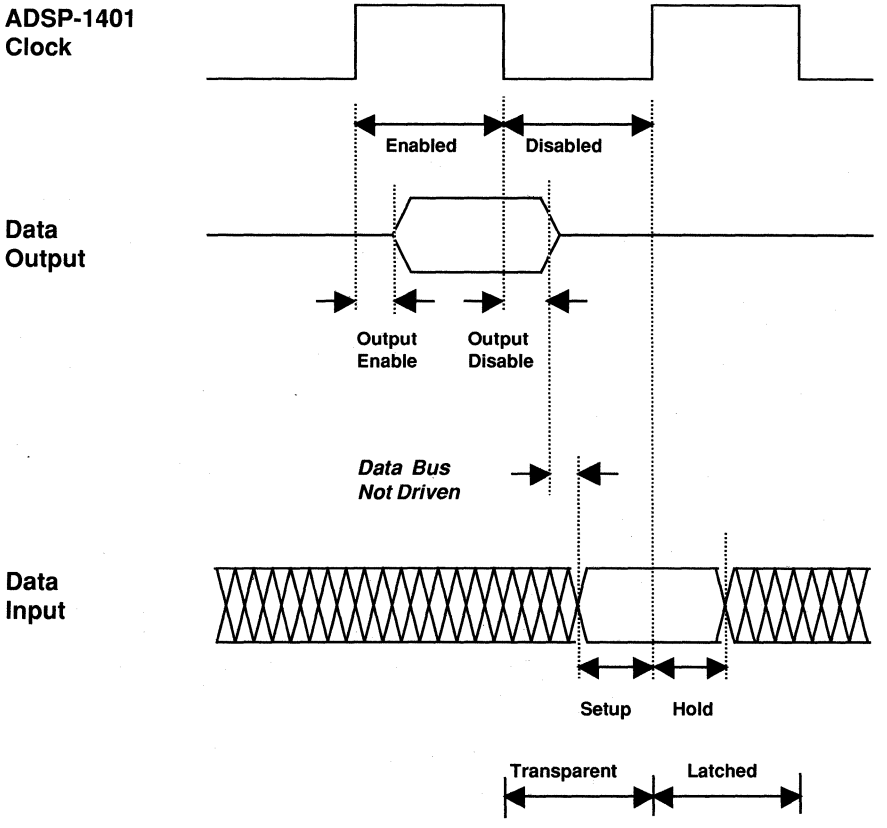


Figure 2.3 Data Output Followed by Data Input

2.5 ADDER AND WIDTH CONTROL

The adder allows the ADSP-1401 to calculate relative jump addresses by adding offset values to the program counter. When the ADSP-1401 is executing a relative jump instruction, it has already incremented the program counter; therefore, the number of instructions from the address of the relative jump instruction to the jump address is the offset value plus one.

The ADSP-1401 reads offsets at the data port. You can program the ADSP-1401 to read the least significant 8 bits or 12 bits, or all 16 bits. Offsets fewer than 16 bits wide are automatically sign-extended before being added to the program counter address. Sign extension repeats the most significant bit to extend the value to 16 bits, as shown in Figure 2.4.

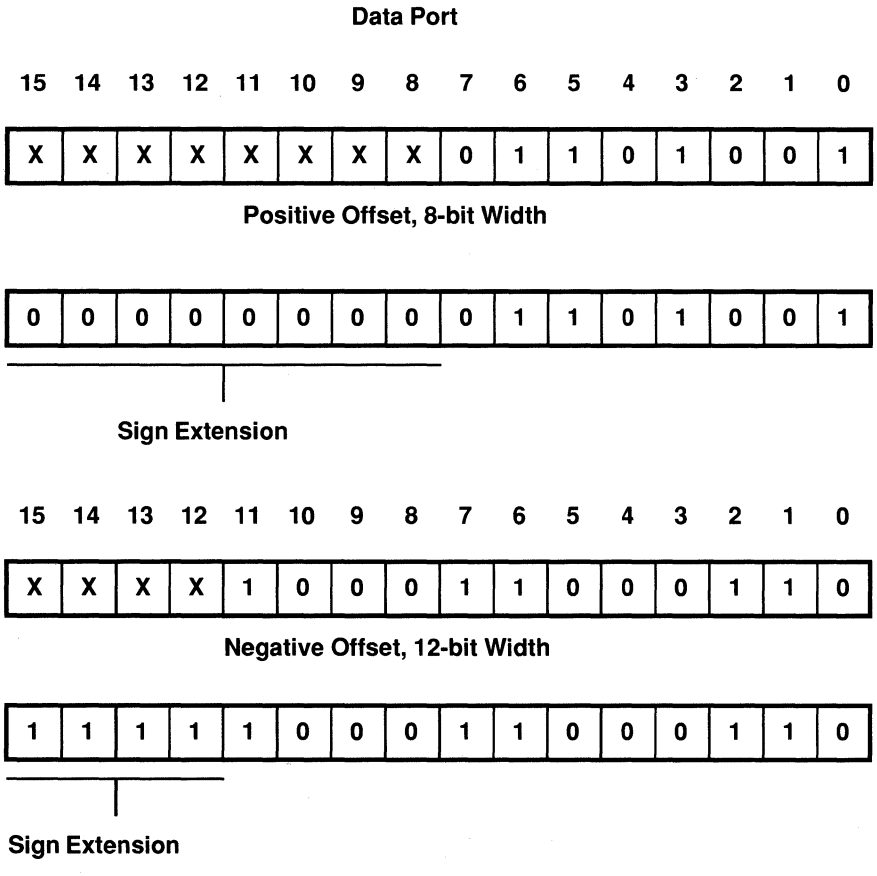


Figure 2.4 Sign Extension

2 Internal Architecture

2.6 INTERNAL RAM AND STACK POINTERS

The internal RAM of the ADSP-1401 consists of 64 16-bit words. A typical RAM configuration is shown in Figure 2.5. The RAM can be divided into the following spaces:

- *Subroutine Stack:* The subroutine stack stores the return address needed to return from a subroutine or interrupt service routine automatically. You can also save counter, register, status and pointer values on the subroutine stack. The subroutine stack pointer (SSP) is a 6-bit register that contains the address of the most recently written stack location (the next stack location to be read). The subroutine stack grows in the direction of higher memory addresses. A subroutine stack push first increments the SSP, then writes the stack location. A pop reads the stack location (without erasing it), then decrements the SSP.

- *Register Stack:* The register stack comprises two other stacks: the local stack and the global stack. The local stack registers store jump addresses for each level of subroutine nesting, whereas the global stack registers store jump addresses accessible from any point in the program. The local stack pointer (LSP) contains the 6-bit address of the most recently written local register (the next register to be read), and the global stack pointer (GSP) contains the 6-bit address of the most recently written global register. Jump instructions that access the addresses stored on the register stack include two bits of opcode that select one of the top four registers. Both the local and global stacks grow in the direction of lower memory addresses. A register stack push first decrements the pointer (LSP or GSP), then writes the stack location. A pop reads the stack location (without erasing it), then increments the pointer.

- *Indirect Address Space:* The indirect address space stores jump addresses that are accessed by their location in the internal RAM. In indirect jump instructions, the RAM is addressed by the six least significant bits of the data port (D_{5-0}).

The reset operation automatically initializes the SSP to location 0. You must initialize the GSP and LSP explicitly. Normally, the LSP is set to a lower address than the GSP, because the local stack must grow to accommodate several levels of nested subroutines, whereas the global stack can often be fixed at the outset of the program. Indirect addresses reside in high RAM (addresses greater than the global stack addresses).

Because the subroutine stack grows toward higher memory and the register stack grows toward lower memory, there is a potential for a stack collision. The Stack Limit Register (SLR) is a 4-bit register that defines the boundary between the two stacks. The ADSP-1401 generates an internal interrupt when it detects the overflow of either the subroutine stack or register stack. See *Stack Overflow and Underflow Interrupt* in Chapter 4 for more information.

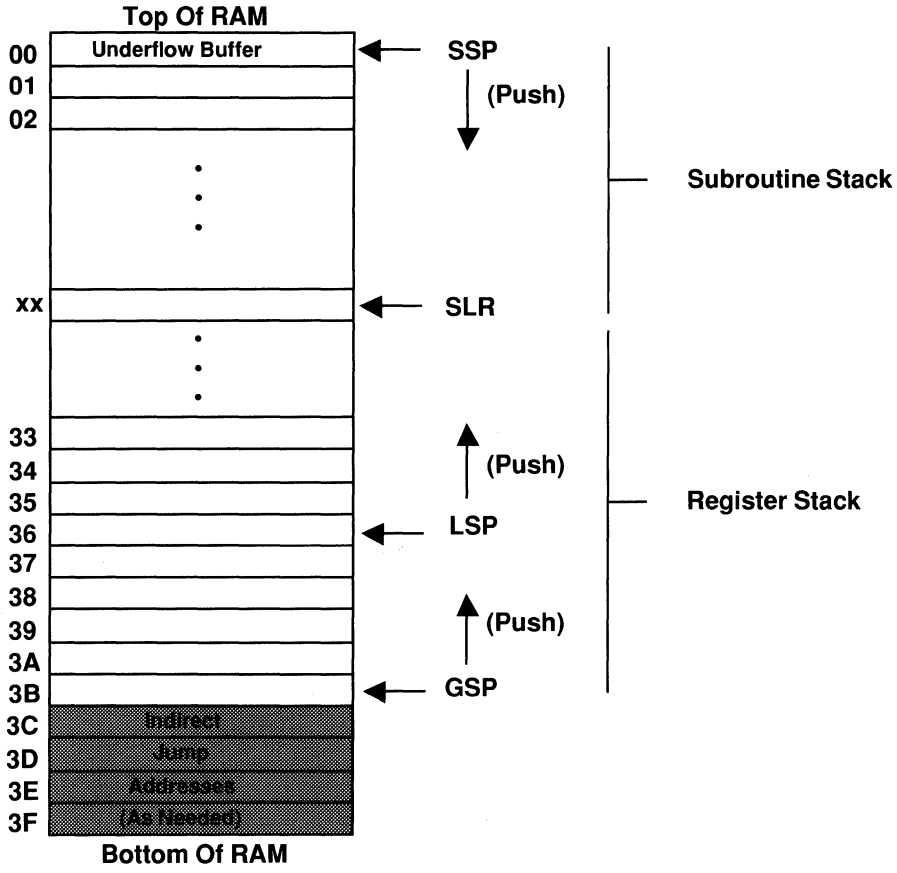


Figure 2.5 Typical Internal RAM Organization

2.7 INTERRUPTS AND INTERRUPT VECTOR FILE

The ADSP-1401 processes up to eight interrupts from external sources plus two interrupts generated internally. The interrupt vector file stores ten 16-bit vectors (jump addresses). Each vector is the starting address of an interrupt service routine for one of the ten interrupts. The ADSP-1401 outputs the vector on the address port in the cycle following the recognition of the interrupt.

2 Internal Architecture

The ADSP-1401 receives external interrupts through four level-sensitive inputs ($EXIR_{4-1}$) which are time-multiplexed to generate eight internal interrupt signals. Interrupts IR_{8-5} are normally latched on the $EXIR_{4-1}$ pins during clock HI, and interrupts IR_{4-1} are normally latched on the $EXIR_{4-1}$ pins during clock LO. Interrupt timing is shown in Figure 2.6.

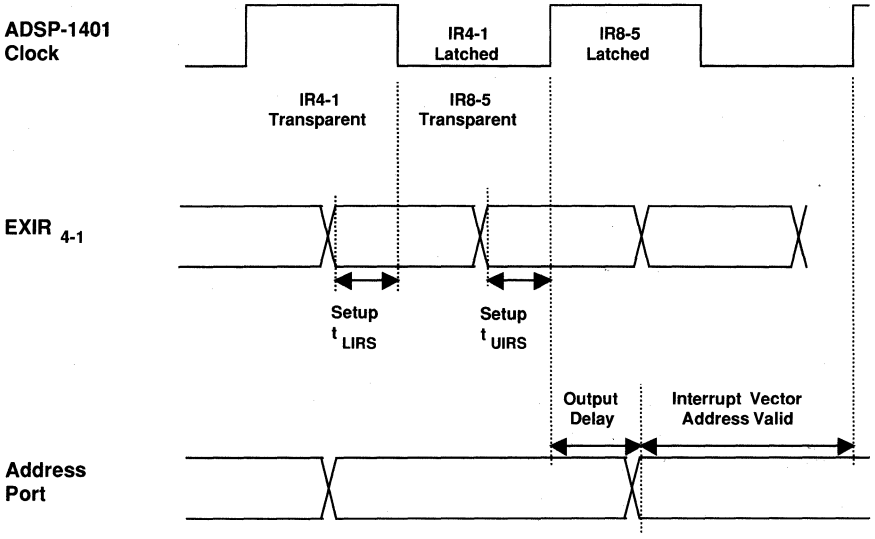


Figure 2.6 External Interrupt Latching

You can enable or disable the processing of all interrupts, or mask interrupts individually. The ADSP-1401 still latches an interrupt while it is disabled or masked; if you re-enable or unmask the latched interrupt, the ADSP-1401 will process it.

The interrupt vector file is loaded from the data port; you can also read interrupt vectors at the data port. The Interrupt Vector Pointer (IVP) is a 4-bit register that contains the location of the next interrupt vector to be loaded or read. You load or read a specific vector by moving the IVP to the vector file location and performing the appropriate instruction.

An interrupting device can remove the interrupt signal after the interrupt has been latched. An interrupt remains latched until cleared by one of several instructions, usually in the interrupt service routine. You can also operate interrupts in a transparent mode, in which the latches are bypassed. In this mode, the interrupting device must maintain the interrupt signal until the interrupt is recognized. The transparent mode makes it possible to cancel an interrupt signal before the next clock edge to prevent interrupt servicing. The latching mode is determined by a bit in the status register; see *Status Register*, below.

Support of more than four interrupts requires an external two-to-one multiplexer, shown in Figure 2.7, to input the interrupt signals to the ADSP-1401 at the correct times. One to four interrupt sources can be connected directly to the $EXIR_{4-1}$ pins. These interrupts should conform to the timing for IR_{8-5}, the higher priority interrupts, and you must also mask interrupts IR_{4-1}. Any unused interrupt inputs (external multiplexer inputs or $EXIR_{4-1}$) should be masked and preferably grounded as well, to prevent them from being activated.

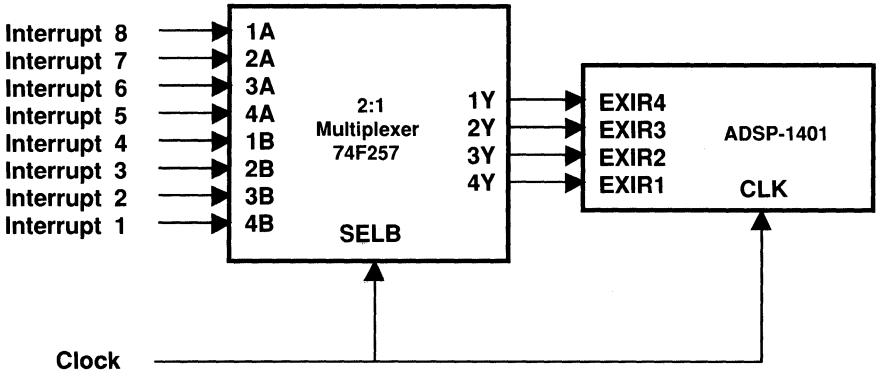


Figure 2.7 Supporting Five to Eight External Interrupts

The internally generated interrupt signals are IR_9, which indicates a stack overflow or underflow, and IR_0, which indicates a counter underflow. These interrupts are described in Chapter 4, *Interrupt Processing*.

2.8 EVENT COUNTERS

The four counters of the ADSP-1401 (C_{3-0}) store 16-bit twos-complement values. These decrement-only counters keep track of events and implement programming loops. Several instructions use the sign bit (most significant bit) of a counter as a condition for executing a particular operation.

The ADSP-1401 always stores the sign bit of the most recently decremented counter (the value *before* decrement) in the status register; see *Status Register*, below. You can condition certain instructions on this stored sign bit. The sign bit of a counter can also generate the internal IR_0 interrupt; this interrupt is provided primarily to end a download operation (see *Writeable Control Store* in Chapter 5). See *Counters* in Chapter 3 for more information on how to use counters to implement loops.

2.9 FLAG

The ADSP-1401 latches the Flag input during clock HI. Several instructions use the Flag input (or its complement) as a condition for executing a particular operation. See *Flag* in Chapter 3 for more information.

2 Internal Architecture

2.10 TRAP/TRISTATE/RESET

The Trap/Tristate/Reset (TTR) input performs three functions:

- *Trap:* The Trap signal is an asynchronous, nonmaskable interrupt. It usually indicates a system emergency, such as a power failure; you can also use it to implement a cache (see *Trap* in Chapter 4). If the TTR input is asserted (HI) during clock LO and deasserted (LO) during clock HI, a Trap signal is generated. The Trap interrupt is serviced by the same routine as IR_9; the IR_9 vector is output a specified time after the assertion of the Trap signal. Therefore, if the Trap signal is used, the routine must determine whether the interrupt source is the internal IR_9 interrupt or the Trap signal.

- *Tristate:* The Tristate signal places the address port in the high-impedance state a specified time after its assertion. This signal allows another device to address the microcode memory without otherwise affecting the ADSP-1401 operation. If the TTR input is HI during clock HI and LO during clock LO (follows the clock signal), the address port is placed in the high impedance state, regardless of the executed instruction.

- *Reset:* The Reset signal initiates a reset operation, which places the ADSP-1401 in a known state. If the TTR input is asserted (HI) during both clock HI and clock LO, the ADSP-1401 initiates a reset operation. The TTR input must remain HI for two more cycles for the reset operation to complete.

The timing of the three signals is compared in Figure 2.8.

The Tristate signal can be active during clock LO only a maximum time of t_{TSOV} from the falling clock edge; otherwise, the Trap signal is activated. Similiarly, the Trap signal can be active during clock HI only a maximum time of t_{TROV} from the rising clock edge without activating Tristate. Figure 2.9 shows a time-multiplexing circuit that coordinates the Trap, Tristate and Reset signals to the TTR input. This circuit ensures that the Trap and Tristate signals are distinct. If both the Trap and Tristate signals are used, they cannot be asserted in the same cycle, because the ADSP-1401 recognizes this combination as a Reset signal.

Internal Architecture 2

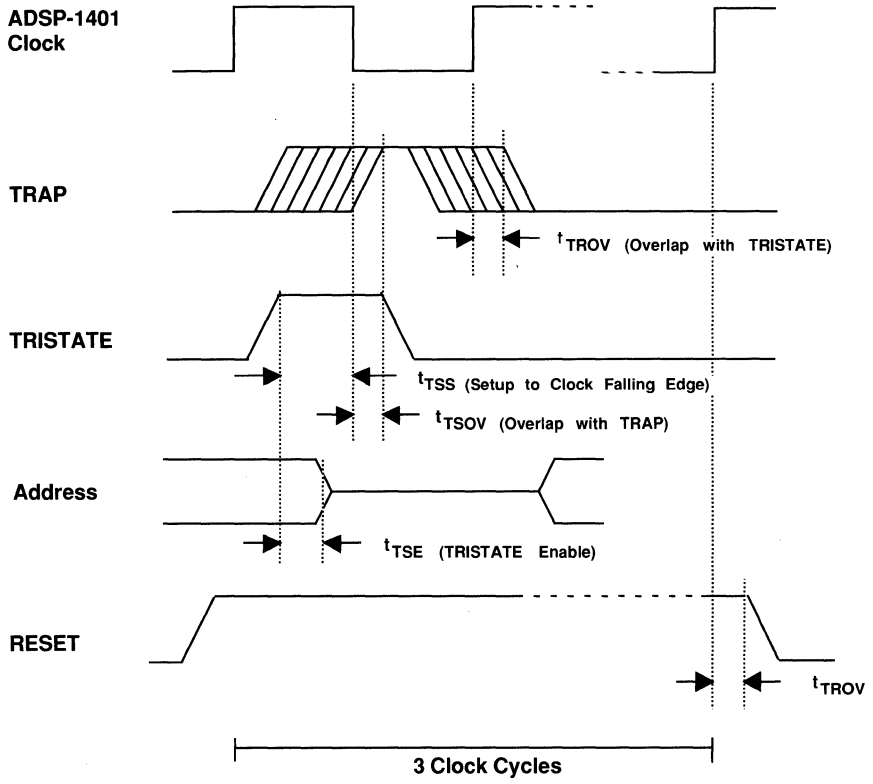


Figure 2.8 Timing for Trap, Tristate, and Reset

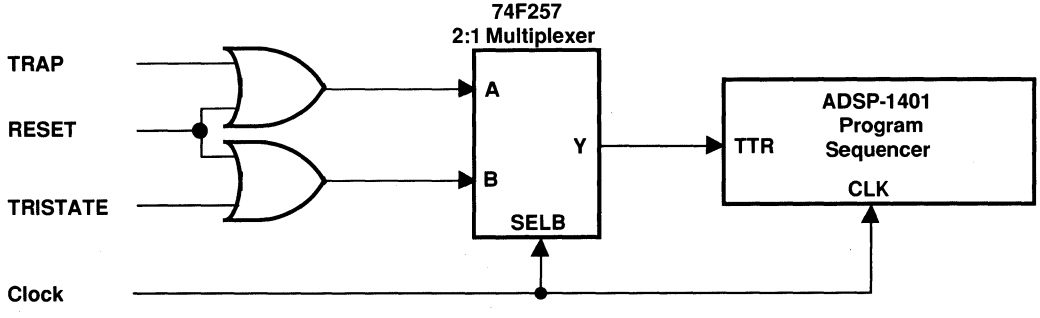


Figure 2.9 TTR Multiplexing Circuit

2 Internal Architecture

2.11 ADDRESS MULTIPLEXER AND ADDRESS PORT

The ADSP-1401 outputs the address of the next microcode instruction on its 16-bit address port (pins Y_{15-0}). Addresses are output during clock HI and latched for the duration of clock LO, as shown in Figure 2.10. No external latches are needed between the address port and the microcode address inputs.

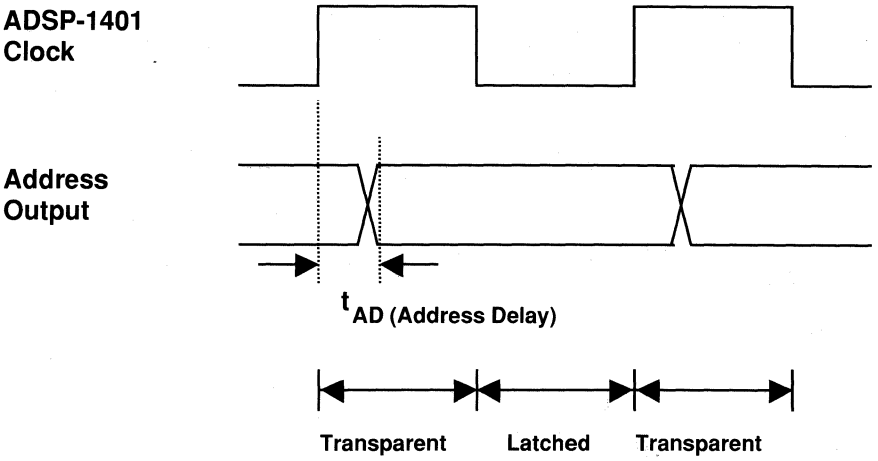


Figure 2.10 Address Output

The address multiplexer selects the address from one of four sources in response to the given instruction:

- *Program Counter:* For sequential addresses
- *Adder:* For absolute or relative jump addresses
- *Interrupt Vector File:* For interrupt service routine addresses
- *Internal RAM:* For indirect jump addresses, local and global jump addresses, and subroutine return addresses

The tristate drivers of the address port are always active unless placed in the high-impedance state through either software (the IDLE instruction) or hardware (the TTR pin) control. For multitasking and context-switching applications, in which other devices must access microcode memory, placing the address drivers of the ADSP-1401 in the high-impedance state effectively removes the device from the address bus.

2.12 STATUS REGISTER

The status register, shown in Figure 2.11, contains 16 bits (SR_{15-0}) that control various operating modes of the ADSP-1401 according to Table 2.2. You can set

Internal Architecture 2

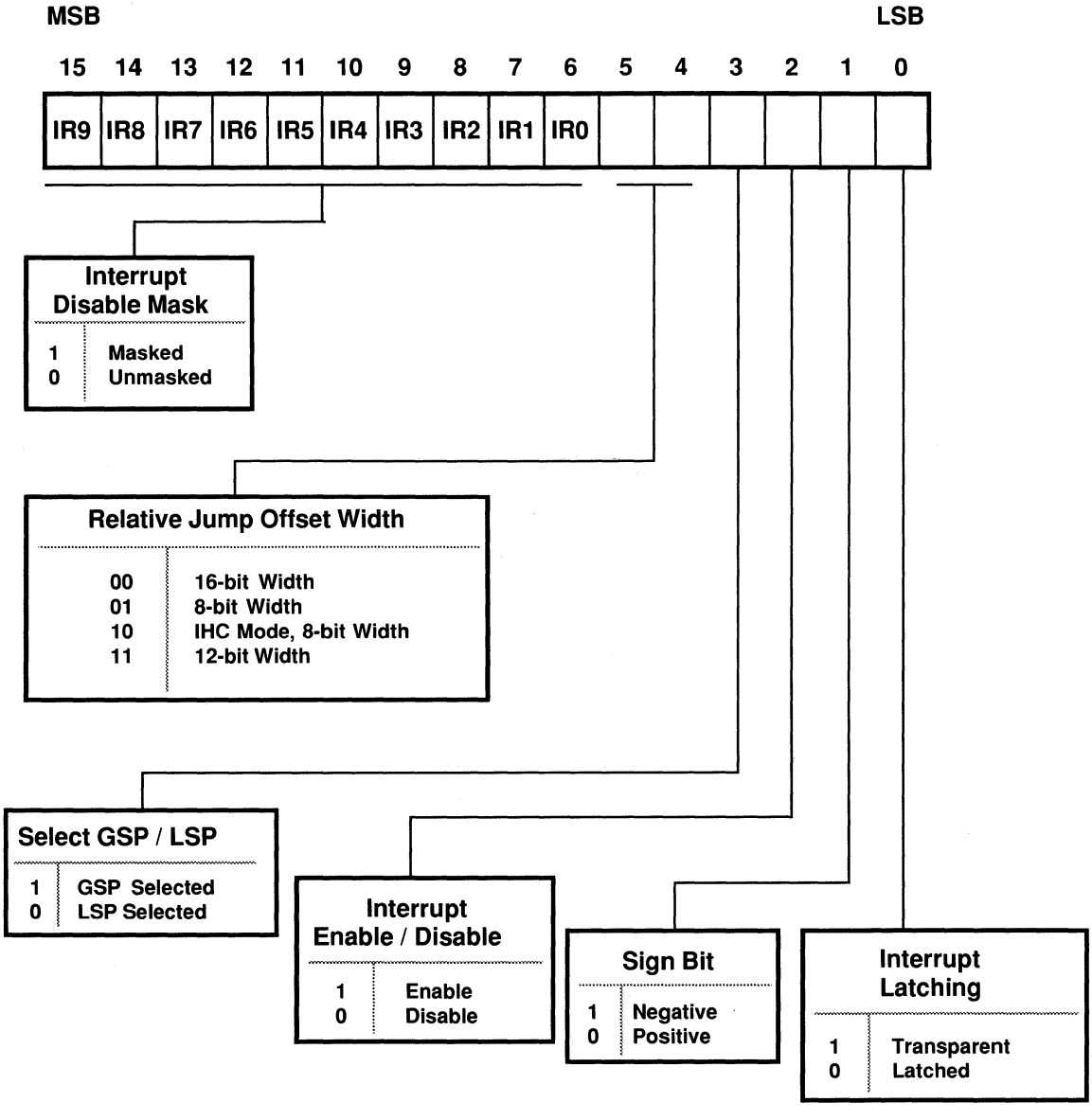


Figure 2.11 Status Register

and clear all bits at once or individually, using dedicated instructions. The ADSP-1401 also writes the SR_1 bit during the execution of instructions that decrement counters. You can load or read the entire status register from the data port. You can also save and restore the status register to and from the subroutine stack. The reset operation clears the entire status register.

SR BIT	*DESCRIPTION*
0	*Transparent Interrupt Selection:* This bit, if set, disables the latching of external interrupts (IR_{8-1}). Transparent interrupts require the interrupt source to maintain the interrupt signal until the service routine acknowledges the recognition of the interrupt. If this bit is cleared, IR_{8-5} are latched during clock HI and IR_{4-1} are latched during clock LO.
1	*Sign Bit:* This bit normally contains the value of the sign bit of the most recently decremented counter (value *before* decrement). It can also be set and cleared using dedicated instructions.
2	*Interrupt Enable:* This bit, if set, enables the processing of all unmasked interrupts. If this bit is cleared, processing of all interrupts, masked and unmasked, is disabled. External interrupts are still latched, unless SR_0 is set.
3	*Register Stack Pointer Selection:* If this bit is set, all register stack instructions affect the global stack. If this bit is cleared, all register stack instructions affect the local stack.
5 - 4	*Relative Jump Offset Width and Instruction Hold Control Mode Selection:* These bits select the width of jump address offsets input at the data port and may also place the ADSP-1401 in the Instruction Hold Control mode, in which the instruction in the instruction latch is repeated under control of the IR_1 interrupt input.
15 - 6	*Interrupt Disable Mask:* These ten mask bits correspond to the ten interrupts. (SR_{15} is paired with IR_9, SR_{14} with IR_8, and so forth.) If the mask bit is set, processing of the corresponding interrupt is disabled, although the interrupt signal is still latched. If the mask bit is cleared, the interrupt is enabled and disabled by the SR_2 bit.

Table 2.2 Status Register Bit Definition

Jumps 3

3.1 INTRODUCTION

Normally, the ADSP-1401 provides sequential execution of microcode by incrementing the current address in the program counter. A jump is a deviation from sequential execution. A jump instruction outputs an address that is different from the next sequential address and reloads the program counter with this new address.

Jumps are used for several purposes, including:

- *Conditional Branches:* A conditional jump creates a branch in the program flow based on a specified condition (the ADSP-1401 provides four conditions). If the condition is true, the ADSP-1401 skips to another section of the program; if the condition is false, the ADSP-1401 continues to execute instructions sequentially. The ADSP-1401 also provides for a three-way branch, in which two conditions are tested instead of one.

- *Subroutine Calls:* A jump to a subroutine executes a block of instructions in one section of the program and then resumes sequential execution. Before the ADSP-1401 performs a subroutine jump, it pushes the address of the next sequential instruction on the subroutine stack. When the subroutine execution completes, the ADSP-1401 pops the subroutine stack and program execution continues from the return address.

- *Loops:* A jump to a previously executed instruction repeats the execution of one or more instructions. The ADSP-1401 includes four event counters that you can use to set the number of loop iterations.

- *Interrupt Servicing:* A jump to an interrupt service routine is similar to a subroutine call in that the ADSP-1401 stores the next sequential address for the return jump. However, the jump is initiated by an interrupt signal rather than a software instruction. Interrupts are explained fully in Chapter 4, *Interrupt Processing*.

These operations are illustrated in Figure 3.1. Chapter 6 provides complete information on the specific instructions that perform these operations.

3 Jumps

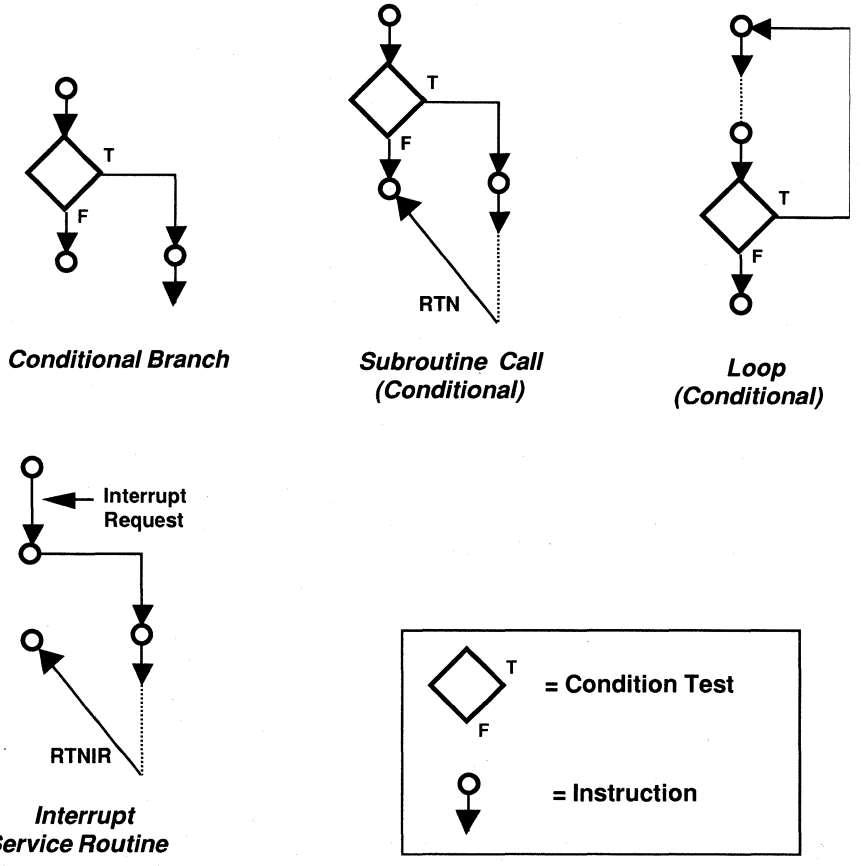


Figure 3.1 Branches, Subroutines, Loops, and Service Routines

The ADSP-1401 obtains jump addresses from several sources:

- *Data Port:* The jump address is usually given in a data field supplied with the jump instruction in microcode. The ADSP-1401 reads the address on its data port and can output this address unchanged (absolute addressing) or add it to the program counter value first (relative addressing).
- *Indirect Address Space:* An external source, usually microcode, provides an address on the six least significant bits of the data port. This 6-bit address selects a location in the indirect address space that contains a preloaded jump address.

- *Interrupt Vector File:* On the cycle following the detection of an interrupt, the ADSP-1401 outputs the appropriate interrupt vector on the address port.

- *Register Stack:* Several jump instructions access an address stored in the local stack or global stack. The stack location is specified relative to the location of the LSP or GSP.

- *Subroutine Stack:* Subroutines and interrupt service routines require jumps to return to their calling routine. To execute a return from a subroutine or interrupt service routine, the ADSP-1401 pops the return address from the subroutine stack.

3.2 CONDITIONS

Conditional jump instructions contain two bits of opcode that are used to select one of four jump conditions. These conditions are often used to control loop iteration and subroutine execution, as well as conditional branching.

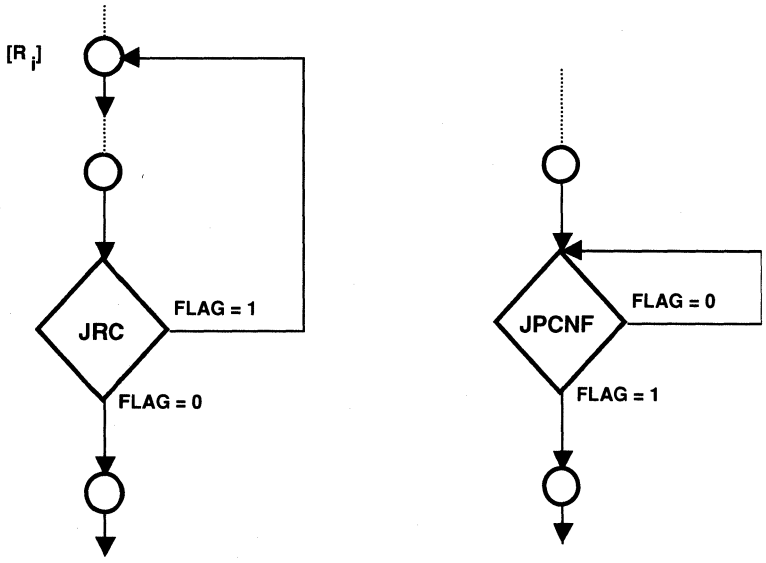
- FLAG - Execute jump if Flag input is HI.
- NOT FLAG - Execute jump if Flag input is LO.
- SIGN - Execute jump if sign bit of specified counter is HI.
- UNCONDITIONAL - Execute jump always

3.2.1 Flag Input

The Flag input signal or its complement (Not Flag) can be selected as a jump condition. Through the Flag input, an external condition can control the execution of a jump instruction.

Figure 3.2 shows two kinds of Flag-controlled loops. In the first example, the jump to the top of the loop (JRC instruction to the address in R_i) is conditioned on the Flag signal, and the loop repeats as long as the external device asserts the Flag input. In the second example, the JPCNF instruction creates a one-instruction loop that repeats the same microcode instruction for as long as the Flag input is inactive. (The JPCOF performs the same function for as long as the Flag input is active.) The Flag input can thus be used by an external device to control the repetition of an instruction. This type of loop is useful for reconciling timing differences.

3 Jumps



Loop While Flag = 1 **Loop Until Flag = 1**

Figure 3.2 Loops Using the Flag Input

When IR_0 (counter underflow interrupt) is not masked, the Flag input setup time is greater than for when IR_0 is masked, as shown in Figure 3.3. In the worst-case timing, a Flag input leads to a counter underflow, and extra time is needed to process the IR_0 interrupt after the underflow is detected. See the current ADSP-1401 Data Sheet for timing requirements. The IR_0 interrupt is described in *Counter Underflow Interrupt*, in Chapter 4.

3.2.2 Counters

The ADSP-1401 has four decrement-only event counters (C_{3-0}), which store 16-bit twos-complement values. The sign bit of any counter can be selected as a condition for several jump instructions; a negative sign bit (HI) represents a true Sign condition. Jump instructions conditioned on a counter sign bit can control the iteration of program loops. The sign bit of a counter (value before decrement) determines whether the loop repeats or ends. If the loop repeats, the counter is decremented.

Before it decrements a counter, the ADSP-1401 writes the counter's sign bit to SR_1. Thus, SR_1 always contains the sign bit (before decrement) of the most recently decremented counter. The Sign condition is tested in one of two ways, depending on the type of instruction. If the instruction decrements a specific

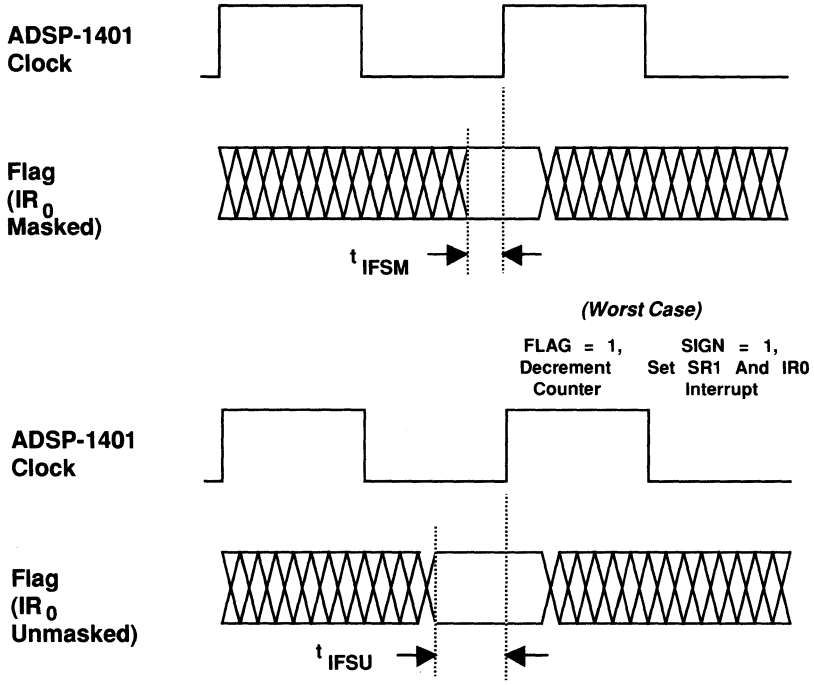


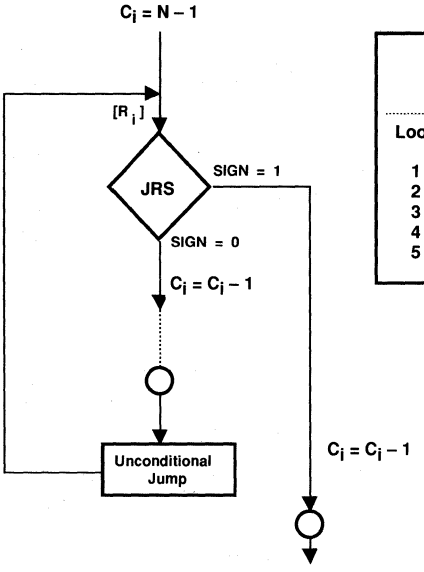
Figure 3.3 Flag Setup Time With and Without IR_0 Masked

counter (conditionally or unconditionally), the sign bit of that counter (before decrement) is tested. If the instruction does not affect a counter, the SR_1 bit is tested. The exception is the IFCDEC instruction; if the Sign condition is selected for this instruction, SR_1 is tested even though the instruction decrements C_0. This instruction incurs a delay when used with the counter underflow interrupt, IR_0; see *Counter Underflow Interrupt* in Chapter 4.

You can load counters from the data port. You can also save them to and restore them from the subroutine stack. The value you must load into a counter to execute a loop depends on the action performed by the instruction and the location of the conditional jump instruction within the loop (top or bottom). For example, in Figure 3.4 the Sign condition is tested at the top of the loop. If the condition is

Figure 3.5 shows another way to construct a loop. The Sign condition is tested at the bottom of the loop. If the condition is true, the JRS instruction executes a jump to the top of the loop (address stored in R_i). Otherwise, the instruction that follows the JRS instruction is executed, ending the loop. In either case, C_i is decremented. To iterate this loop N times, the counter must be loaded with a value 2^{15}+N–2, a negative value, to perform the jump *while* the Sign condition is true.

3 Jumps

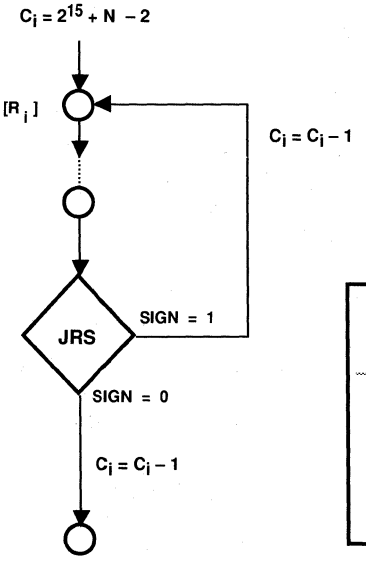


Example: N = 5

5 – 1 = H#0004

Loop	C_i (before decrement)
1	0004
2	0003
3	0002
4	0001
5	0000

Figure 3.4 Loop Until Sign Bit = 1



Example: N = 5

2^{15} + 5 – 2 = H#8003

Loop	C_i (Before Decrement)
1	8003
2	8002
3	8001
4	8000
5	7FFF

Figure 3.5 Loop While Sign Bit = 1

Jumps 3

Examples using other instructions to create loops and count events are described in *Using the Counters of the ADSP-1401 Program Sequencer for Loop and Event Counting*. The use of the counter underflow interrupt (IR_0) is explained in this application note, as well as in Chapter 4, *Interrupt Processing*.

3.3 ADDRESS SOURCES

The ADSP-1401 supports both direct and indirect addressing in its jump instructions. A direct address points to a microcode memory location. An indirect address points to a location in the indirect address space in the internal RAM of the ADSP-1401; this location in turn contains an address that points to a microcode memory location. The register stack is a third source of jump addresses. These addresses are accessed through either the GSP or LSP.

3.3.1 Direct Addressing

Direct addressing requires input from the data port. The ADSP-1401 can perform two types of direct jumps: an absolute jump, which interprets the given data value as the jump address, and a relative jump, which interprets the given data value as an offset from the program counter address. To perform an absolute jump, the ADSP-1401 passes the data port value through the width control block and adder to the address port unchanged. For a relative jump, the ADSP-1401 adds the data port value to the program counter value, and outputs the sum on the address port. Because the program counter contains the address of the next instruction, the jump distance from the relative jump instruction is one instruction more than the offset value. If the twos-complement offset is negative, however, the jump distance is one less than the offset value. In Figure 3.6, an offset of H#0024 results in a jump from address H#0123 to H#0148, whereas an offset of H#FFDC (the twos complement of H#0024) results in a jump from H#0123 to H#0100.

The ADSP-1401 recognizes three offset widths: 16, 12, and 8 bits. This feature conserves data bits in microcode for applications that do not require relative jump offset values larger than the selected data width can represent. Two status register bits select the offset width. The ADSP-1401 ignores the unused data bits when reading an offset from the data port.

3.3.2 Indirect Addressing

Indirect jump addresses are stored in the indirect address space (in the internal RAM). A 6-bit address provided at the data port with an indirect jump instruction specifies the RAM location. In principle, an indirect jump address can be stored in any RAM location. In practice, however, the indirect address space is usually located in the highest memory to eliminate the possibility of an indirect jump address being overwritten by a stack operation.

3 Jumps

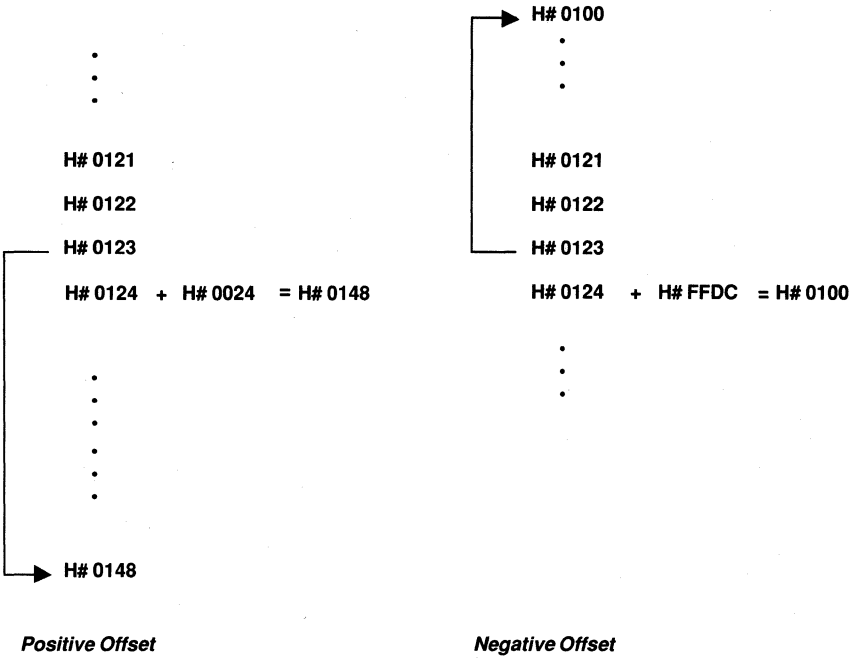


Figure 3.6 Positive and Negative Offsets

3.3.3 Register Stack

The register stack consists of two stacks: the local stack and the global stack. Both of these stacks provide storage for jump addresses. The subroutine stack can be used for this purpose, but the SSP must be tracked so that subroutine nesting levels are not lost. The register stack does not have this constraint. The register stack pointers (LSP and GSP) can be moved as needed to access stack contents.

The local stack stores jump addresses used by particular subroutines. The global stack stores jump addresses that can be accessed from any subroutine, as well as the main program. For example, the global stack might contain a jump address used by several interrupt service routines. Sharing jump addresses on the global stack avoids the duplication of jump addresses in the local stack and conserves the code required to push the jump addresses on the local stack.

A single set of register stack instructions applies to both the local stack and the global stack. A status register bit determines which stack a register stack instruction affects. Certain jump instructions can access four registers of the selected stack as jump addresses. These registers are referenced by their location relative to the LSP or GSP; for example, R_2 is the register at the location that is

two greater than the LSP or GSP location. To access four other registers, you simply move the stack pointer.

The local stack can grow for each level of subroutine nesting. The number of registers depends on the number of local stack pushes performed by the subroutine; however, only the top four registers can be accessed without moving the LSP. Figure 3.7 shows an example of a local stack after three subroutine calls. The LSP pointed to location H#1D in the main program. Subroutine A pushed the stack twice, subroutine B pushed the stack six times, and subroutine C pushed the stack twice. Before the return from subroutine C is executed, the subroutine must remove its registers from the stack, either by adding two to the LSP (requires one instruction) or by popping the stack twice (requires two instructions).

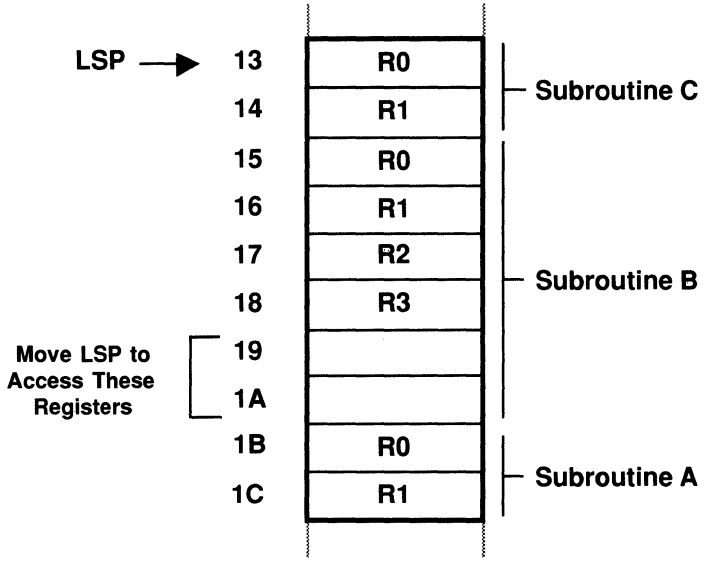


Figure 3.7 Local Stack After Three Subroutine Calls

Global stack registers can be loaded at the outset of the program or as the need arises. Moving the GSP does not affect the stack contents; therefore, the GSP can be reloaded to access different sets of four registers. You can relocate the GSP temporarily by saving it on the subroutine stack and later restoring it (using dedicated instructions).

If a register stack push violates the upper limit of the register stack defined by the Stack Limit Register (SLR), a stack overflow interrupt is generated. See *Stack Overflow and Underflow Interrupt* in Chapter 4 for details.

3 Jumps

3.4 SUBROUTINES

The execution of a subroutine requires two jumps: one jump to the first instruction of the subroutine, and a return jump from the end of the subroutine to the instruction following the subroutine call. To store the return address, the ADSP-1401 automatically pushes the return address on the subroutine stack before jumping to the subroutine. To return to the calling routine, the RTN instruction pops the return address from the subroutine stack to the address port and the program counter. The timings of a subroutine jump and return are shown in Figure 3.8.

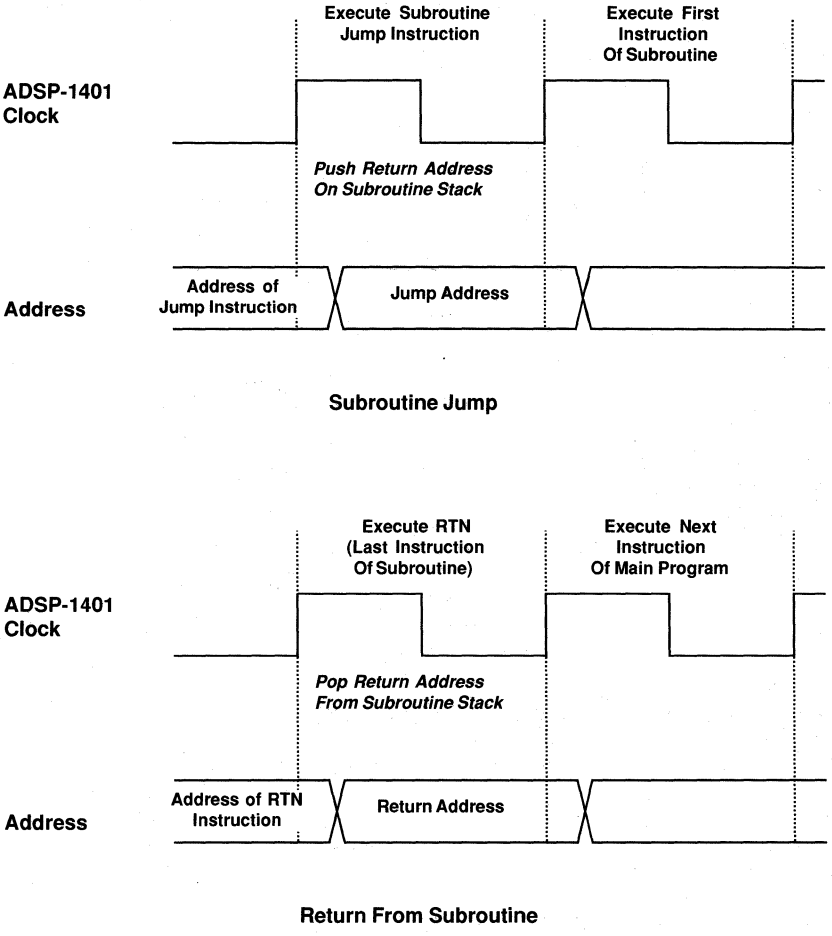


Figure 3.8 Subroutine Jump and Return

The subroutine stack also provides storage for register, counter, and status values. In general, a subroutine should be executable from any point in the program; therefore, it should not alter any data stored in the ADSP-1401 registers, counters, and so forth, unless that is the intent of the subroutine. It is a good idea for a subroutine to save the data of any locations it modifies so that it can restore them before returning to the calling routine. The status register, any of the four counters, and the GSP can be pushed on the subroutine stack. At the end of the subroutine, the subroutine stack must be popped in the reverse order, as shown in Figure 3.9. Note that a pop does not erase a stack location

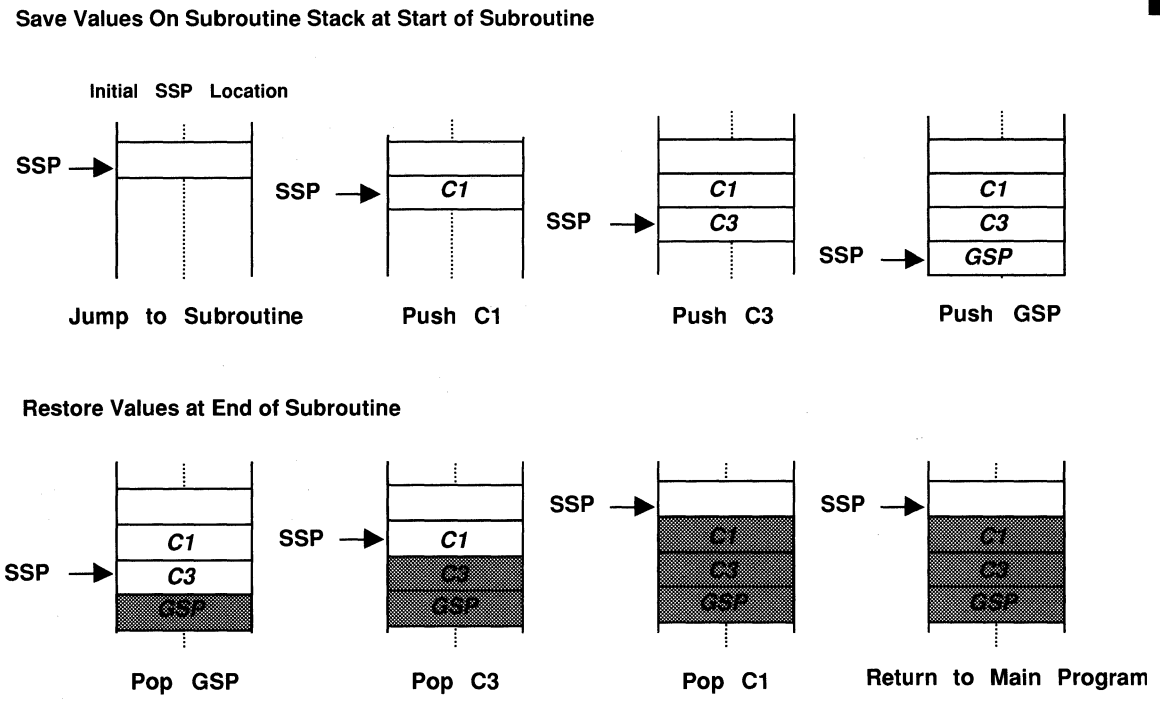


Figure 3.9 Restoring Registers and Counters

If a subroutine stack push violates the upper limit of the subroutine stack defined by the Stack Limit Register (SLR), a stack overflow interrupt is generated. See *Stack Overflow and Underflow Interrupt* in Chapter 4 for details.

A subroutine is nested when it is called during the execution of another subroutine. Figure 3.10 shows an example of the contents of the internal RAM with three levels of subroutine nesting.

- Four indirect addresses and two global jump addresses are loaded in the main program.
- The first subroutine jump pushes a return address on the subroutine stack. Counters C_1 and C_3, the GSP, and the status register are pushed on the subroutine stack in the first subroutine, to preserve their values for the main program. After the GSP has been saved, the subroutine pushes two jump addresses on the global stack, as well as two jump addresses on the local stack.
- The second subroutine jump pushes a return address on the subroutine stack. Counters C_1 and C_2, and the GSP are pushed on the subroutine stack to preserve their values for the first subroutine. After the GSP has been saved, the second subroutine pushes three jump addresses on the global stack, on top of those already saved by the first subroutine. These global stack addresses can be used by any subsequently nested subroutines. The second subroutine uses no local registers.
- The third subroutine jump pushes a return address on the subroutine stack. The status register is pushed on the subroutine stack to preserve its value for the second subroutine. The third subroutine does not alter the global stack and can access the same global registers as the second subroutine. In this subroutine, the LSP is selected and four jump addresses are pushed on the local stack.

The first and second subroutines, which save the GSP before pushing the global stack, can remove their jump addresses from the global stack by simply restoring the GSP value. Local jump addresses must be removed by incrementing the LSP.

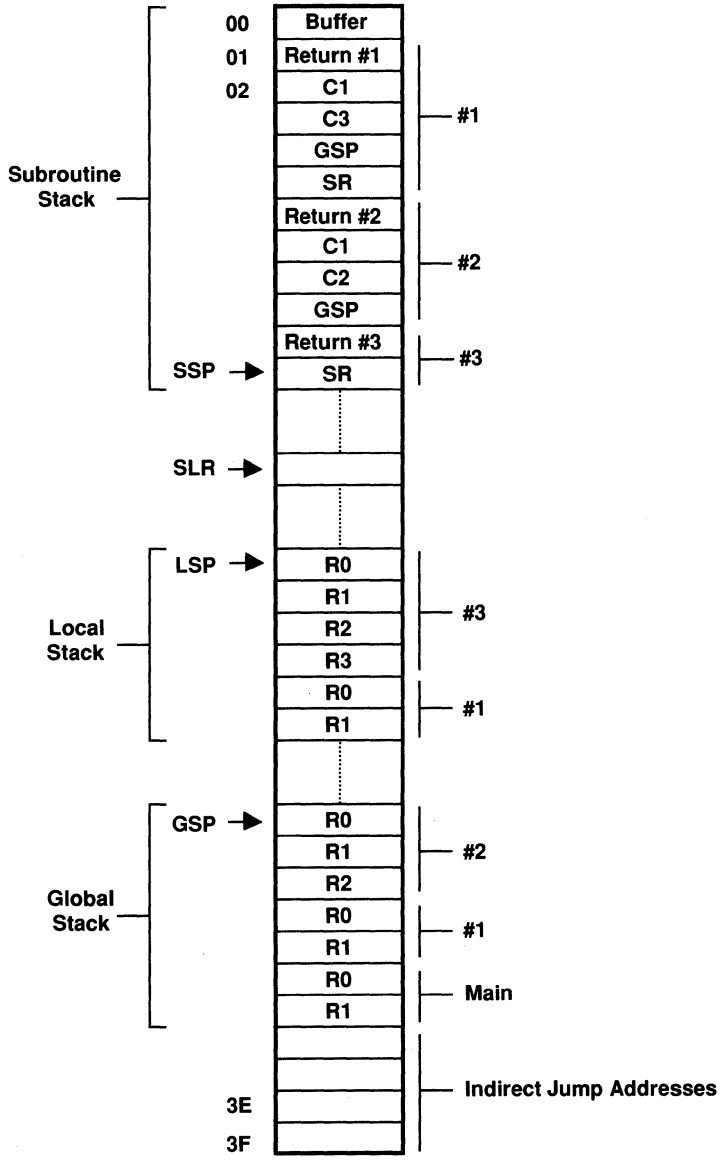


Figure 3.10 Internal RAM With Three Levels of Nesting

Interrupt Processing 4

4.1 INTRODUCTION

The ADSP-1401 supports up to eight external interrupt sources and two additional interrupts generated internally. Each interrupt is paired with an interrupt vector stored in the interrupt vector file; the interrupt vector points to the service routine for that interrupt.

You can load or read interrupt vectors at the data port by first loading the 4-bit Interrupt Vector Pointer (IVP) with the vector location (from the data port, using a dedicated instruction), then executing the RDIV or WRIV instruction. You should disable interrupt processing before writing or reading the vector file. The ADSP-1401 automatically increments the IVP after each write or read to facilitate loading or dumping of the entire vector file.

When more than one interrupt occurs in the same cycle, interrupt priority determines the order in which the interrupts are serviced. The ADSP-1401 assigns priority according to the interrupt number; IR_9 has the highest priority (gets serviced first) and IR_0 has the lowest priority (gets serviced last).

4.2 MASKING AND ENABLING

Interrupt processing can be enabled and disabled through two methods:

- The interrupt mask in the status register selectively enables and disables each interrupt. The mask bits SR_{15-6} correspond to the interrupt signals IR_{9-0}, respectively. If the mask bit is set, the corresponding interrupt signal is masked out (disabled), although it is still latched and will be processed when the interrupt is unmasked. Any combination of mask bits can be set or cleared in a single instruction; the data port value selects the bits to be set or the bits to be cleared.

- SR_2 enables or disables all interrupts. If SR_2 is not set, interrupt vector output is inhibited, although interrupts are still latched and will be processed when interrupts become enabled.

If an interrupt is pending (latched) when the interrupt is unmasked or re-enabled, the interrupt vector is output in the cycle after the unmasking or re-enabling instruction. Therefore, the instruction that follows the unmasking or enabling instruction is executed before the first instruction of the interrupt service routine, as shown in Figure 4.1.

4 Interrupt Processing

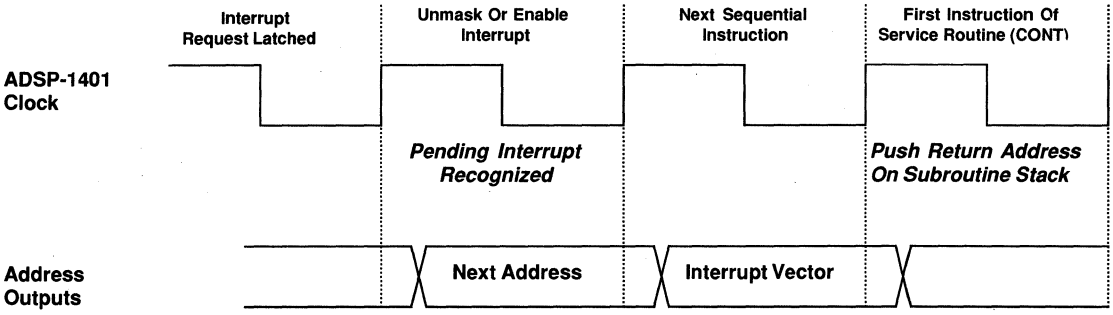


Figure 4.1 One-Cycle Latency After Enabling Interrupts

If an interrupt is latched in the same cycle that an instruction masks or disables this interrupt, it is recognized and its vector is output in the next cycle. An interrupt that occurs in the cycle after the masking or disabling instruction is latched but not processed until the interrupt is unmasked or re-enabled.

4.3 INTERRUPT SERVICE ROUTINES

Interrupt service routines are similar to subroutines. They require a jump to the first instruction of the routine and a return jump at the end of the routine. However, service routines have the following differences:

- Jumps to service routines are automatically performed by the ADSP-1401 in response to an active interrupt rather than a jump instruction.

- The ADSP-1401 pushes the return address on the subroutine stack in the cycle after the interrupt vector is output (during the first instruction of the interrupt service routine), whereas a subroutine return address is pushed in the same cycle that the jump address is output. The instruction being executed when the vector is output may be accessing the internal RAM, so the stack cannot be pushed until the instruction completes. The first instruction of any service routine must always be a CONT instruction to allow the ADSP-1401 to save the return address.

- When the ADSP-1401 begins processing an interrupt, it asserts an internal Interrupt In Progress (IRIP) signal that prevents other active interrupt signals from being processed. A return from an interrupt routine (the RTNIR instruction) not only pops the return address from the subroutine stack, but also clears IRIP to re-enable the processing of pending interrupts and clears the interrupt latch for the current interrupt (external only, IR_{8-1}). The RTNIR instruction does not clear internal interrupts (IR_9 and IR_0); they must be cleared explicitly, using dedicated instructions.

Interrupt service routines and subroutines are compared in Figure 4.2.

Interrupt Processing 4

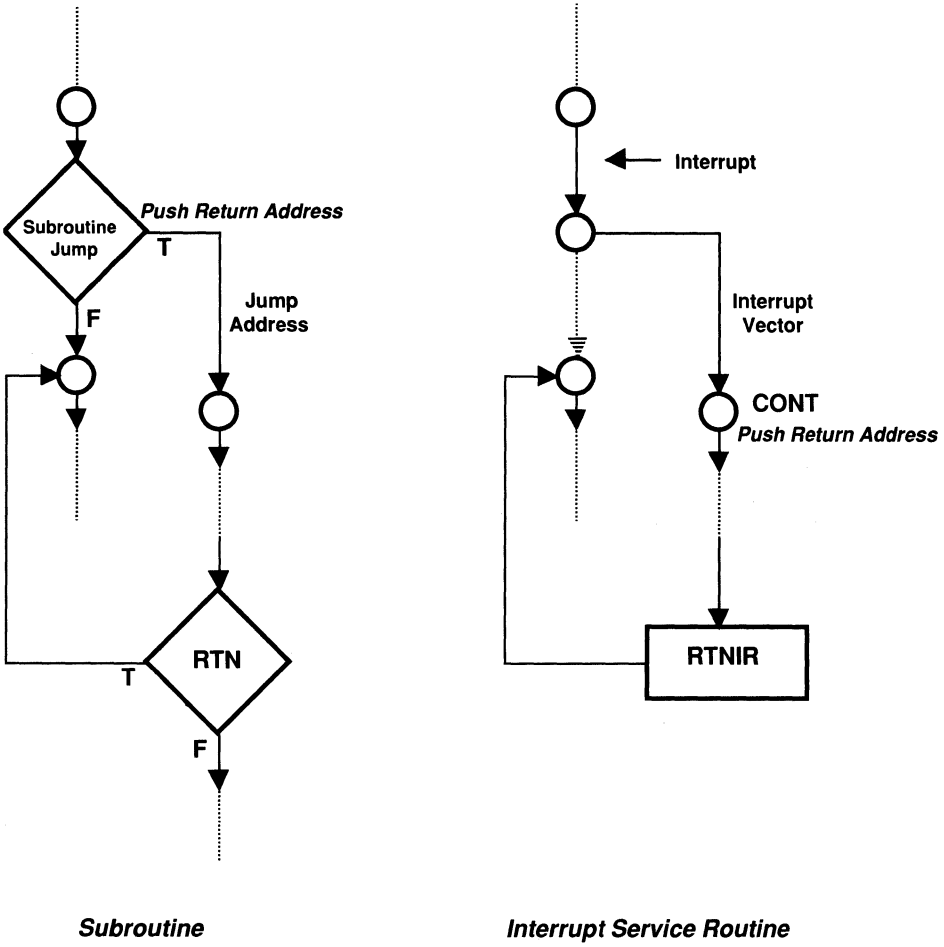


Figure 4.2 Execution of Service Routines and Subroutines

If the IRIP signal is cleared before the end of the service routine, a new service routine can be nested within the current routine. One of two instructions clears the IRIP signal:

- The CCIR (Clear Current External Interrupt) instruction clears the IRIP signal as well as the interrupt latch for the interrupt currently being serviced (external interrupts only).

4 Interrupt Processing

- The CAIR (Clear All External Interrupts) instruction clears the IRIP signal and all external interrupt latches. Internal interrupts (IR_9 and IR_0) are not cleared.

A pending interrupt is recognized in the cycle in which one of these instructions is executed, and its interrupt vector is output on the following cycle. Therefore, the instruction that follows the CCIR instruction or CAIR instruction is executed before the nested service routine begins, as shown in Figure 4.3. In this example, IR_7 and IR_3 occur in the same cycle. IR_7 is processed first. After the CCIR instruction in the IR_7 service routine re-enables interrupt processing, IR_3 is processed.

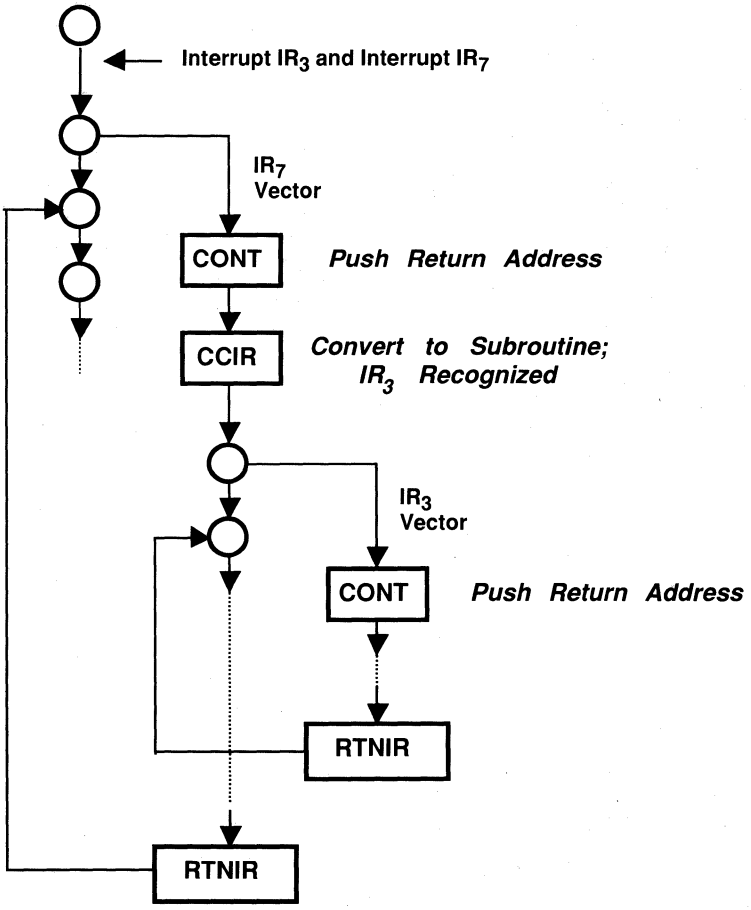


Figure 4.3 Interrupting a Service Routine

After the execution of the CCIR instruction or the CAIR instruction, the service routine becomes identical to a subroutine. It is generally a good idea to convert service routines to subroutines by executing CCIR or CAIR early in the routine for the following reasons:

- Interrupts that occur while IRIP is active cannot be serviced until after IRIP is cleared. A high priority interrupt can be ignored for a potentially indeterminate length of time.

- If you use the nonmaskable Trap interrupt (see *Trap*, below), you cannot rely on the IRIP signal to remain active for the duration of any service routine. The Trap interrupt is not locked out by the IRIP signal. If the Trap input is asserted while IRIP is active, its service routine is nested in the current service routine. The return from the Trap service routine clears the IRIP signal. Because the Trap interrupt can occur at any time, the IRIP signal can be cleared at any time.

If service routines are not nested, a pending interrupt is processed after the return from the current interrupt routine. One instruction from the main program is executed between the RTNIR instruction and the first instruction of the next interrupt routine, because the interrupt vector is output in the cycle after the RTNIR instruction; this sequence of events is shown in Figure 4.4.

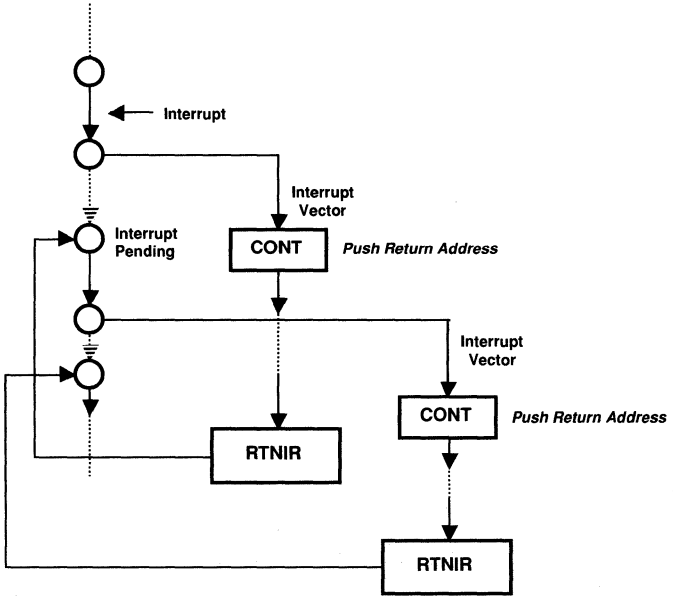


Figure 4.4 Pending Interrupt on Return From Service Routine

4 Interrupt Processing

If an interrupt occurs in the cycle before a jump (a subroutine jump or a return from a subroutine, for example), the interrupt vector displaces the jump address or return address, which is pushed on the subroutine stack. After the return from the interrupt routine, the jump address is output and the jump or return is executed, as shown in Figure 4.5.

4.4 COUNTER UNDERFLOW INTERRUPT

The ADSP-1401 generates an IR_0 interrupt, the lowest priority interrupt, whenever a counter underflows (its sign bit goes HI). This interrupt is used primarily to end a writeable control store download (see *Writeable Control Store* in Chapter 5). When IR_0 is enabled, the Flag input setup time is greater than for when IR_0 is masked; see *Flag Input* in Chapter 3. The application note *Using the Counters of the ADSP-1401 Program Sequencer for Loop and Event Counting* describes the use of IR_0 in implementing loops.

The RTNIR instruction does not clear IR_0. The CLRS instruction clears both the IR_0 interrupt and SR_1, which are both set by a counter underflow. The IR_0 service routine must execute a CLRS instruction before returning; otherwise, the still-active IR_0 signal will reactivate the service routine.

4.5 STACK OVERFLOW AND UNDERFLOW INTERRUPT

A stack overflows when its pointer moves beyond the upper limit of the stack and underflows when its pointer moves beyond the lower limit of the stack. Because the subroutine stack and register stack grow toward each other, an overflow of one stack can cause stack pushes to overwrite data in the other stack. Underflow can result in pops of locations outside the stack that may contain invalid data. IR_9 flags a stack overflow or underflow so that the service routine can take steps to preserve the stack data. You can implement stack paging, described below, using this interrupt.

The ADSP-1401 generates IR_9 whenever one of the events illustrated in Figure 4.6 occurs. Note that only a push or a pop activates the interrupt; incrementing or decrementing the stack pointer does not cause an interrupt.

- *Subroutine Stack Underflow:* The subroutine stack is popped from location zero; the resulting SSP value is –63.

- *Subroutine Stack Overflow:* The subroutine stack is pushed to its upper limit (see *Stack Limit Register*, below).

- *Register Stack Overflow:* The register stack (global stack or local stack) is pushed to its upper limit, which is three locations greater than the upper limit of the subroutine stack (see *Stack Limit Register*, below).

Because all of these conditions indicate a possible error in stack operation and can result in data loss, interrupt IR_9 has the highest priority. The interrupt service routine must determine which of the three events occurred.

Interrupt Processing 4

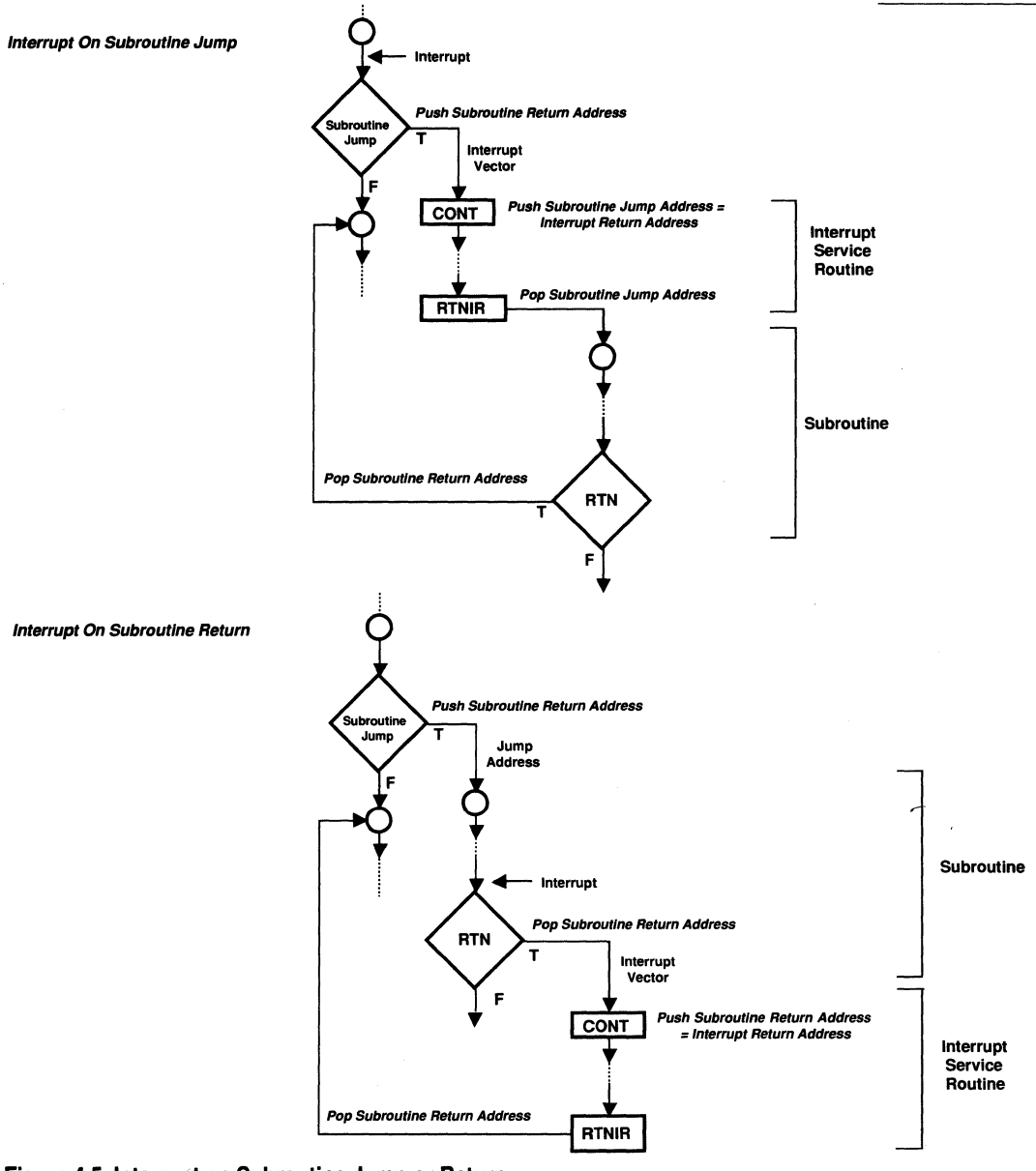


Figure 4.5 Interrupt on Subroutine Jump or Return

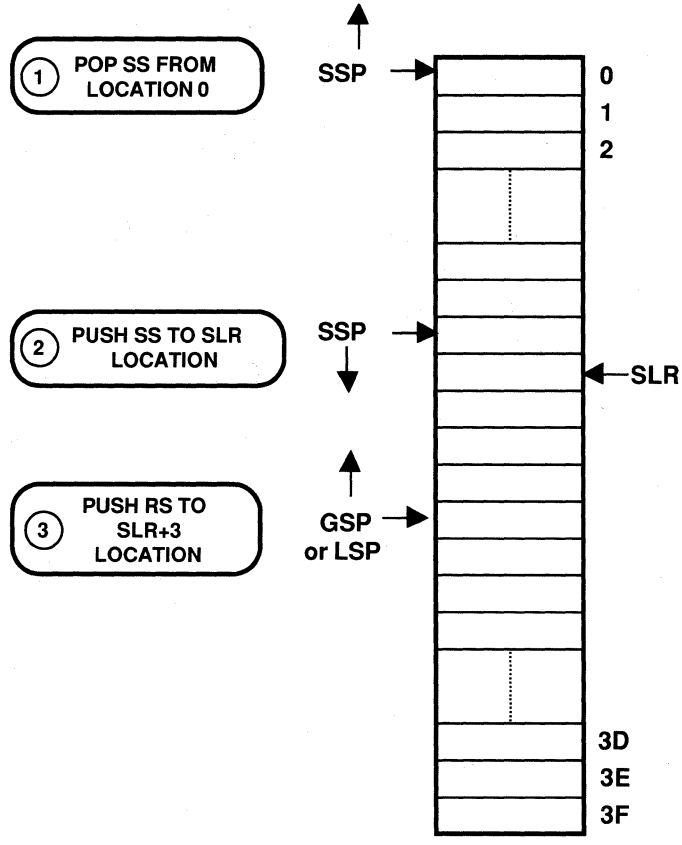


Figure 4.6 Stack Underflow and Overflow

The RTNIR instruction does not clear IR_9. An active IR_9 interrupt remains latched until cleared by the SLRIVP instruction, which also writes both the SLR (described below) and IVP. The IR_9 service routine must execute a SLRIVP instruction before returning; otherwise, the still-active IR_9 signal will reactivate the service routine.

4.5.1 Stack Limit Register

The 4-bit Stack Limit Register (SLR) stores a value that defines the boundary between the subroutine stack and the register stack. After each subroutine stack push, the ADSP-1401 automatically appends two least significant zeros to the

SLR value and compares it to the 6-bit SSP value. After each global or local stack push, the ADSP-1401 automatically appends two least significant ones to the 4-bit SLR value before comparing it to the 6-bit GSP or LSP value. In each case, if the values match a stack overflow (IR_9) interrupt is generated. Thus, the subroutine stack overflows if a push causes the SSP to equal the SLR value, and the register stack overflows if a push causes the GSP or LSP to equal the SLR value + 3. Note that because the SLR contains only the four MSBs of the stack limit, the SLR value must be placed at a four-word boundary.

The three-location buffer between the upper limits of the two stacks protects both stacks from data loss. At the most, three locations can be filled before the first usable (non-CONT) instruction of the IR_9 service routine is executed, as shown in Figure 4.7. One location, on either the subroutine stack or register stack, is loaded by the push that causes the overflow interrupt. Another location is filled by a push on either stack that may be executed in the cycle during which the interrupt vector is output. The third location, on the subroutine stack, is filled by the return address of the IR_9 service routine.

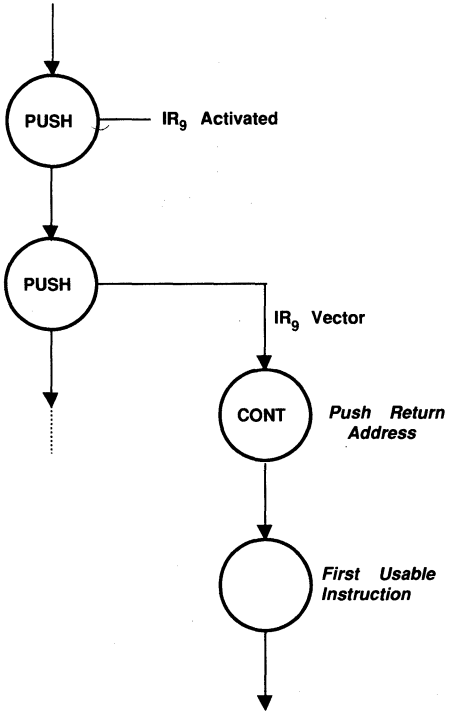


Figure 4.7 Three Pushes On Stack Overflow

4.5.2 Stack Paging

A subroutine stack overflow can occur any time the number of internal RAM locations allocated to the subroutine stack is less than the number of return addresses, registers, counters, and so forth, stored on the stack. This situation usually occurs in programs that employ several subroutine nesting levels. One strategy for handling stack overflow is stack paging. Figure 4.8 is a simplified illustration of the paging operation. A stack paging routine saves the contents of the stack in external memory when the stack overflows, allowing the stack area to be overwritten without losing the stack information. As the stack is popped, it eventually underflows, and the stack paging routine must read back the previous page from external memory. Because both stack overflow and stack underflow activate IR_9, the IR_9 service routine can perform both paging functions.

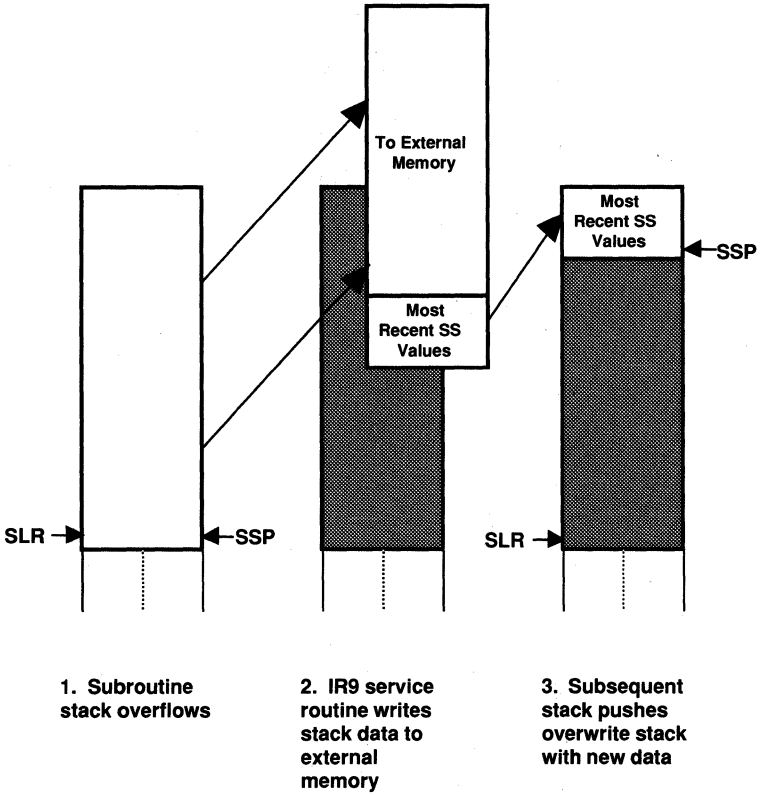


Figure 4.8 Stack Paging

The application note *Stack Paging Expands Internal RAM of the ADSP-1401 Program Sequencer* details the operations involved in stack paging.

4.6 TRAP

The Trap input is one of the three functions of the TTR pin. Trap generates a nonmaskable interrupt that has the highest priority and usually flags a situation requiring immediate attention, such as an imminent power failure or a cache memory page fault. The Trap signal shares the IR_9 interrupt vector with the IR_9 interrupt. In systems that utilize both the Trap function and the stack overflow/underflow interrupt, the IR_9 service routine must determine whether the Trap signal or a stack overflow or underflow caused the interrupt.

The Trap signal is input asynchronously. If the TTR pin is asserted (HI) during clock LO and deasserted (LO) during clock HI, the internal Trap signal is activated and the IR_9 interrupt vector is output a time t_{TRAD} later. The timing of the Trap signal is described in *Trap/Tristate/Reset* in Chapter 2.

The Trap interrupt differs from other interrupts in that the current address (the program counter value, which is address of the instruction being executed when Trap is asserted) is pushed on the subroutine stack rather than the next address (program counter value plus one). Thus, the ADSP-1401 repeats the instruction in which Trap is asserted after the return from the Trap service routine. This feature facilitates the implementation of a cache memory system. If you use the Trap signal for something other than a cache, you should take this feature into consideration.

A cache system consists of a small amount of fast (short access time) memory, called the cache, and a larger amount of slower and less expensive memory. The cache contains copies of some of the locations in the slow memory. When a fetch from one of these locations is requested (called a cache hit), the fetch requires only the fast memory access time. When a fetch from a location not in the cache is requested (called a cache miss or page fault), the cache controller circuitry must copy the memory location from the slow memory to the fast memory before the fetch can be performed. The Trap input is used to flag the cache miss. After the Trap service routine performs the memory transfer, the fetch that caused the cache miss is repeated. Implementations of both instruction and data caches are described in the application note *Implement A Cache Memory In Your Word Slice System.*

When the Trap input is used to signal a cache miss, the ADSP-1401 must output the IR_9 interrupt vector *before* the next cycle, not during the next cycle as for other interrupts, because the cache miss must be resolved and the requested fetch completed before the next instruction fetch. This may require stretching the clock LO period, as shown in Figure 4.9. The service routine vector is output during the stretched clock LO. When the service routine completes, the return address, which is the address that caused the cache miss, is output once again.

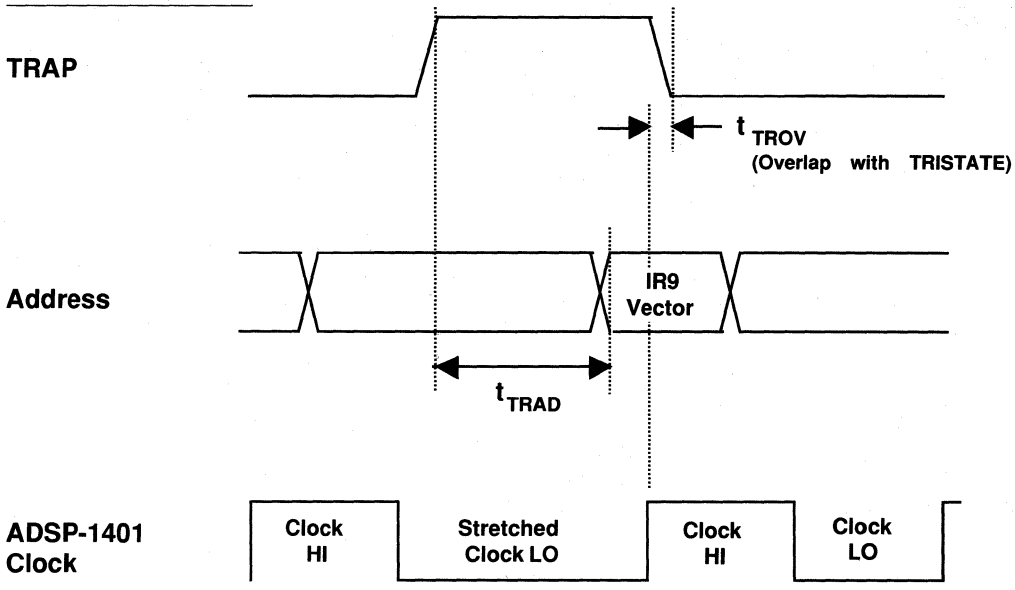


Figure 4.9 Stretching the Clock LO Period

System Interface 5

5.1 INTRODUCTION

The ADSP-1401 receives its instructions from microcode, and it may also receive data and control from microcode or from other devices in the system. It outputs addresses to microcode. These inputs and outputs determine the system interface requirements of the ADSP-1401.

5.2 LOOK-AHEAD PIPELINE

The Look-Ahead pipeline consists of two complementary latching stages. The instruction port and data port (during data input) are latched during clock HI and transparent during clock LO. The address port is latched in just the opposite fashion; it is transparent during clock HI, latched during clock LO. The ADSP-1401 outputs an address to microcode during clock HI; this address is stable because the instruction is latched. When the instruction port becomes transparent during clock LO, the address latch holds the address for the rest of the cycle. The instruction for the next cycle is partially decoded while the latch holds the address for the current cycle, as shown in Figure 5.1.

The Look-Ahead pipeline provides two major advantages:

- No external instruction register or address registers are needed. The ADSP-1401 can connect directly to microcode memory at both input and output.
- Because the next instruction is partially decoded before the start of the next cycle, the address can be output sooner, allowing more time for the memory access. Jump addresses are output in the same cycle that a jump instruction is executed, without the complexity and inefficiency of delayed branches.

5.3 DATA INPUT AND OUTPUT

The ADSP-1401 can transfer data between its data port and most internal locations. Data input operations load values into the internal RAM, registers, counters, and the adder. Data output operations dump internal values for examination.

5 System Interface

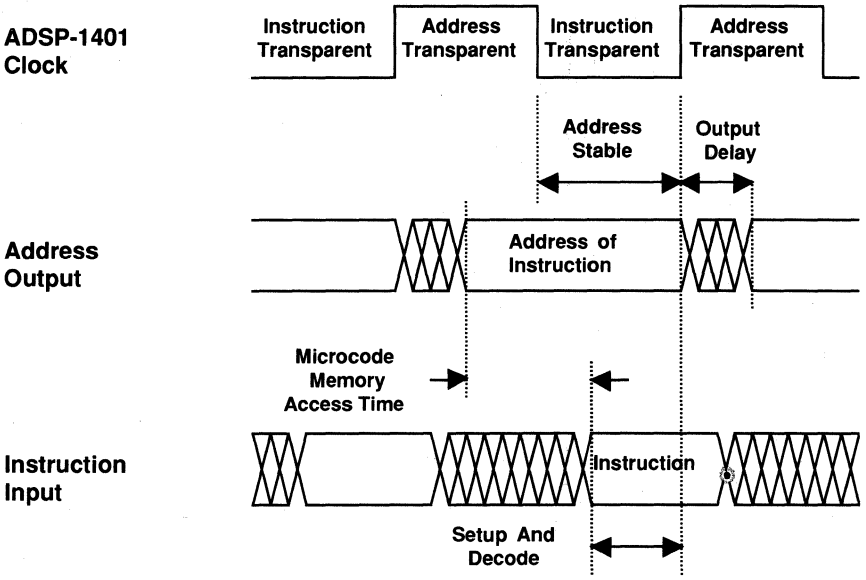


Figure 5.1 Look-Ahead Pipeline Timing

You can load and read all of the following internal locations through the data port using the following instructions:

LOCATION	*TO LOAD*	*TO READ*
Subroutine Stack	PSDSS	PPSSD
Local Stack	PSDRS*	PPRSD*
Global Stack	PSDRS†	PPRSD†
SSP	WRSSP	RDSSP
LSP	WRRSP*	RDRSP*
GSP	WRRSP†	RDRSP†
SLR	SLRIVP	nonreadable
Interrupt Vector File	WRIV	RDIV
IVP	SLRIVP	nonreadable
Counters	WRCNTR	PSCNTR and
PPSSD		
Status Register	WRSR	RDSR

* Local stack selected ($SR_3 = 0$)
† Global stack selected ($SR_3 = 1$)

The SR_3 bit controls the selection of either the LSP or GSP. The SGSP instruction sets SR_3, and the SLSP instruction clears SR_3. The SLRIVP instruction, which loads the SLR and the IVP simultaneously, writes the values on D_{5-2} to the SLR and the values on D_{15-12} to the IVP. Neither the SLR nor the IVP can be transferred to the data port.

The indirect address space can be loaded using any stack pointer (GSP, LSP, or SSP). You can dump the contents of any range of the internal RAM by first setting a stack pointer to the lowest address and then executing the appropriate pop instruction (PPRSD or PPSSD) once for each location.

You can load the entire status register directly from the data port. Also, individual status register bits may be set and cleared through explicit instructions. You can load counters from the data port, but you cannot read them directly. However, you can push a counter on to the subroutine stack and pop the counter value from the stack to the data port.

5.4 INSTRUCTION HOLD CONTROL

In the Instruction Hold Control (IHC) mode, the ADSP-1401 can repeat the instruction in its instruction latch, instead of latching a new instruction, under control of an external signal. The external interrupt $\overline{IR}_1$ is redefined in this mode to provide this control input. The IHC mode is frequently used with the IDLE instruction to suspend the operation of the ADSP-1401, allowing another device to address the same instruction space in microcode. Shared microcode can often reduce microcode memory requirements.

You select the IHC mode by writing the value 10 (binary) to SR_{5-4}, using the IHC instruction. Other settings of SR_{5-4} set the width of offsets used for relative jumps, as listed in Table 5.1. When the IHC mode is selected, only 8-bit-wide offsets are recognized. You can use any of the instructions that set offset width (REL16, REL12, and REL8) to change the values of SR_{5-4} and thereby deselect the IHC mode.

SR_5	SR_4	*Selection*
0	0	16-bit Offset Width
0	1	8-bit Offset Width
1	0	IHC Mode, 8-bit Offset Width
1	1	12-bit Offset Width

Table 5.1 Relative Jump Offset Width Selection

In the IHC mode, an active $\overline{IR}_1$ signal initiates instruction hold. The $\overline{IR}_1$ signal is transparent during clock HI, and latched during clock LO on pin $EXIR_1$. IR_5, which is timeshared with $\overline{IR}_1$ on $EXIR_1$ but latched during clock HI, is unaffected as long as the necessary multiplexing circuit is used (see *Interrupts and Interrupt Vector File* in Chapter 2).

5 System Interface

The instruction that is in the instruction latch when IR_1 is asserted (HI) is repeated until IR_1 is deasserted (LO), as shown in Figure 5.2. Assertion and deassertion of IR_1 must meet the instruction setup time requirement.

Because IR_1 is still latched in IHC mode, you must mask the IR_1 interrupt (set SR_7) to allow the ADSP-1401 to use IR_1 as a control signal without generating a false interrupt. You should mask IR_1 permanently to dedicate it to IHC control, if possible. Interrupts that occur during instruction hold cannot be saved unless IR_1 is dedicated to IHC control. If you use the IR_1 signal for an interrupt as well as for IHC, you must mask it before selecting the IHC mode and unmask it after deselecting the IHC mode. Before unmasking IR_1, however, you must also clear the latch (using the CAIR instruction) because the use of IR_1 as an IHC control sets the IR_1 latch. The CAIR instruction also clears any pending interrupts.

5.5 TRISTATE OUTPUTS

In some applications, such as context switching and multitasking, the microcode is addressed by more than one device. These applications require a means of removing the ADSP-1401 from the microcode address bus temporarily. The ADSP-1401 provides two methods through which you can place its address outputs in the high impedance state: in software, using the IDLE instruction, and in hardware, using the Tristate function of the TTR input. The IDLE instruction suspends internal operation as well, whereas the Tristate input allows the ADSP-1401 to continue performing instructions without outputting an address.

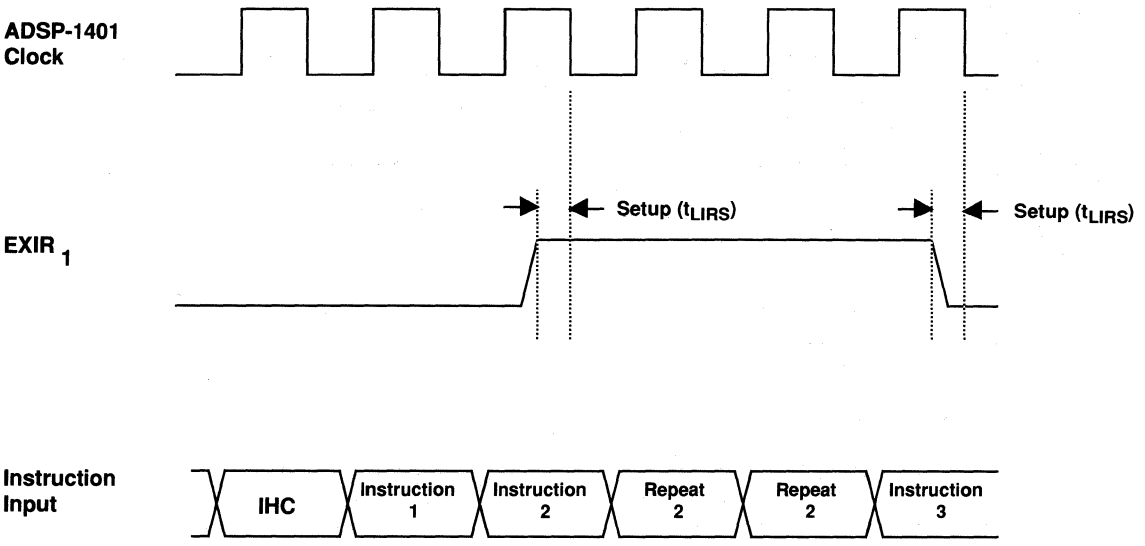


Figure 5.2 Instruction Hold Control Using IR_1

The IDLE instruction holds the program counter at its current value and places the address port in the high impedance state. The IDLE instruction is normally executed repeatedly using the Instruction Hold Control (IHC) mode. This technique puts the device in an idle state in which ADSP-1401 suspends operation temporarily under external control, as described in the application note *Sharing the Output Bus of the ADSP-1401 MicroProgram Sequencer*. When the IHC control signal is removed, the ADSP-1401 continues execution from the instruction that follows the IDLE instruction in microcode.

When the ADSP-1401 is in the idle state, it continues to latch interrupts. Because the address outputs are in the high impedance state, however, no interrupt vector can be output. You should therefore disable or mask interrupts before entering the idle state.

The Tristate function of the TTR pin is activated when the TTR input is HI during clock HI a minimum setup time before the falling clock edge. The TTR input must also go LO within the maximum delay time from the falling clock edge during clock LO, to avoid activating the Trap function. If both the Trap and Tristate signals are used, they cannot be asserted in the same cycle, because the ADSP-1401 recognizes this combination as a Reset signal. See *Trap/Tristate/Reset* in Chapter 2 for a time-multiplexing circuit that coordinates the timing of the Trap, Tristate and Reset signals to the TTR input.

5.6 WRITEABLE CONTROL STORE

A writeable control store is a microcode memory that can be loaded from an external source, usually a host or a DMA circuit. This type of microcode memory provides more flexibility than a fixed-program (ROM-based) microcode memory, because you can change the microcode easily for debugging and for design changes. The application note *Implement a Writeable Control Store In Your Word Slice System* describes several writeable control store configurations.

A download to a writeable control store requires the host to perform a series of writes to the microcode memory. The ADSP-1401, through its WCS (Writeable Control Store) instruction, simplifies the download process by providing microcode addresses in sequence from a given starting address. The host provides the starting address, the microcode instructions to load into memory, and the signals to control the data transfer, as shown in Figure 5.3.

A download is initiated when the ADSP-1401 latches the WCS instruction at its instruction port; it also reads the value at its data port in the same cycle to use as the starting (lowest) address. Once the instruction and data are latched, the ADSP-1401 does not read the instruction port or data port again until the download is terminated. The WCS instruction loads the data port value into the program counter, outputs the same value on the address port, and then performs the following actions repetitively:

5 System Interface

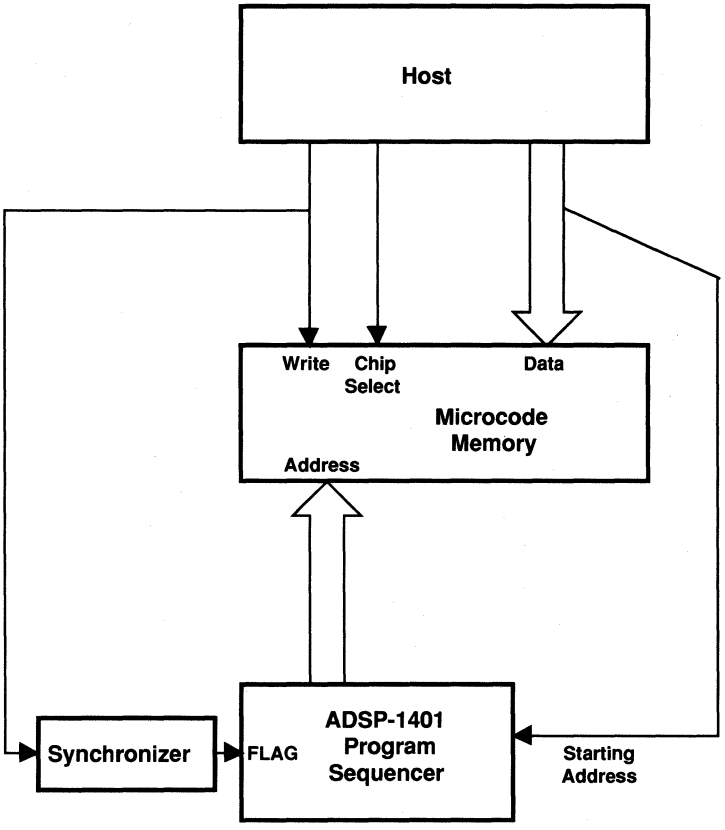


Figure 5.3 Download From Host to Writeable Control Store

- Waits for the Flag input to be asserted
- Increments the program counter
- Decrements the C_0 counter
- Outputs the program counter address to microcode memory

The Flag input should not be asserted until after the WCS instruction has been recognized at the instruction latch; otherwise, the first address will be incremented before being output. The timing of the download operation is shown in Figure 5.4. In this example, the Flag signal is timed so that one instruction is written every two ADSP-1401 clock cycles.

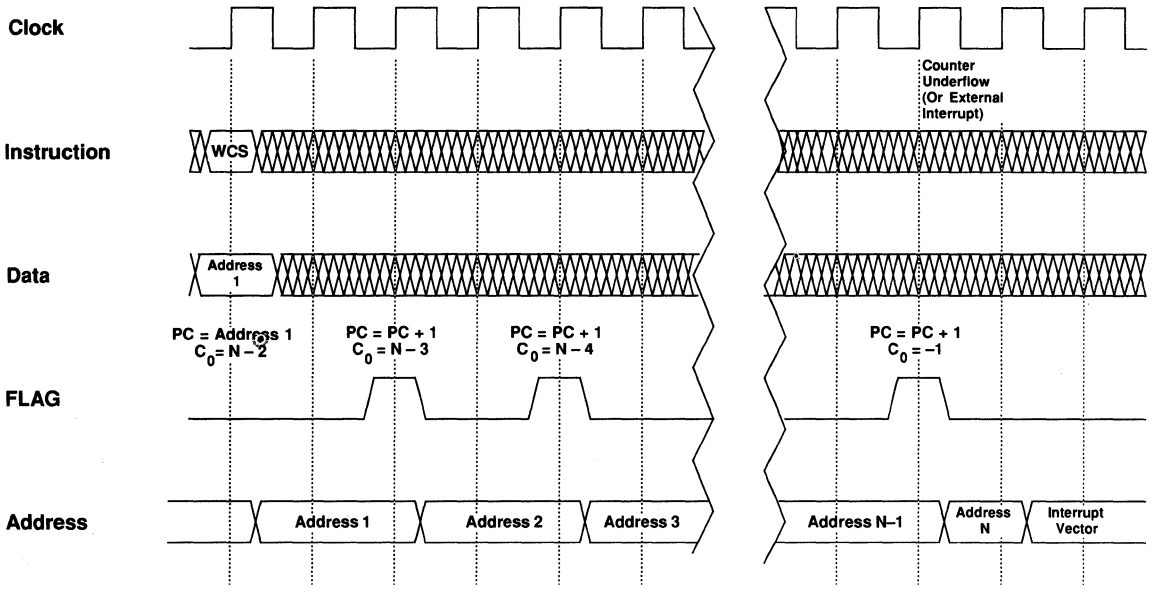


Figure 5.4 WCS Timing

The Flag input serves as a handshake to control how frequently the ADSP-1401 outputs an address. In many systems, the host is not able to output data and control signals as quickly as the ADSP-1401 can output addresses. By asserting the Flag input only after it has output the data, the host guarantees that the next address is not output too soon. If the host can perform writes at the clock frequency of the ADSP-1401, the handshake is unnecessary and the Flag input should be held HI for the duration of the download.

The Flag signal should be synchronized to the ADSP-1401 clock signal to guarantee meeting setup and hold times; a synchronizing circuit is shown in Figure 5.5. This circuit is a digital one-shot that generates a synchronized Flag signal from a negative-true input signal. This input signal must be asserted for at least one ADSP-1401 clock cycle to ensure its recognition.

You can terminate a download through one of three methods:

- Counter underflow interrupt (IR_0)
- One of the eight external interrupts (IR_{8-1})
- A hardware reset (Assertion of TTR for three cycles)

5 System Interface

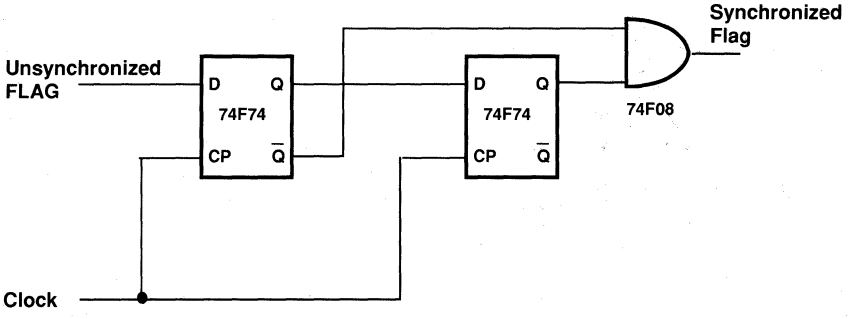


Figure 5.5 Flag Synchronization For WCS

A counter underflow interrupt or external interrupt ends the download by causing the ADSP-1401 to jump to an interrupt vector address and begin executing instructions from that location. The interrupt vector must therefore point to the location in microcode from which the ADSP-1401 is to being sequencing instructions after the download. You should meet the following requirements to account for the fact that no interrupt routine is being executed:

- The interrupt vector location must contain a CONT instruction, because the ADSP-1401 saves the return address on the first instruction of any interrupt routine. The CONT instruction does nothing but output the next address.
- The SSP should be decremented. Because the return address that the ADSP-1401 pushes on the subroutine stack will not be used, it should be removed from the stack.
- The CCIR or CAIR instruction must be executed to clear the interrupt latch and IRIP signal.

The WCS instruction decrements the C_0 counter each time it increments the program counter. If IR_0 is enabled, the counter underflow interrupt will eventually go active, ending the instruction. The number to preload into C_0 is two less than the number of locations to be downloaded, because the ADSP-1401 outputs the first address before it first decrements the counter and the last address in the cycle in which the counter underflows. The interrupt vector is output in the cycle after the counter underflows.

An external interrupt can also end the download. This interrupt may originate from circuitry that decodes the last address or the end of the data file. If you use an external interrupt to end the download, you should either mask the IR_0 interrupt or preload C_0 with a value greater than the number of downloaded instructions so that it never underflows. Note that the ADSP-1401 outputs one address after it detects the interrupt, because the interrupt vector is not output until the following cycle.

To end the download with a reset operation, you must assert the TTR input for three consecutive cycles. The first address output by the ADSP-1401 after a reset is H#0000. Because the first instruction executed by the ADSP-1401 after a reset must be a CONT instruction, you must load the CONT instruction into location H#0000 if you end the download with a reset. For more information about the reset operation, see *Reset*, below.

5.7 RESET

To activate the Reset function, the TTR pin must be HI for both phases of one cycle; to complete the Reset operation, the input must continue to be held HI for two more complete cycles. The Reset function of the TTR pin initializes the ADSP-1401 to the state described in Table 5.2. The address bus is *not* placed in the high-impedance state during the reset operation.

As mentioned in the previous section, the Trap and Tristate signals must not be asserted in the same cycle in systems that use all three functions of the TTR pin. See *Trap/Tristate/Reset* in Chapter 2 for a time-multiplexing circuit that coordinates the timing of the Trap, Tristate and Reset signals to the TTR input.

On powerup, the TTR pin should be held HI before the clock is started and for three cycles after the clock is started to reset the ADSP-1401. Note that the counters, the internal RAM, and the interrupt vector file are unaffected by the Reset operation and are therefore in an indeterminate state after powerup. H#0000 is the first address output on the address port. The first instruction executed by the ADSP-1401 must be a CONT, so location H#0000 must contain a CONT instruction.

LOCATION	*RESET STATUS*
Program Counter	H#0000
Subroutine Stack Pointer (SSP)	H#00
Local Stack Pointer (LSP)	Invalid
Global Stack Pointer (GSP)	Invalid
Stack Limit Register (SLR)	10 00xx (H#20)
Internal RAM Locations	No Change
Counters	No Change
Interrupt Latches (IR_{8-1})	Cleared
Interrupt Vector File	No Change
Interrupt Vector Pointer (IVP)	Invalid
Interrupt Mask (SR_{15-6})	00 0000 0000 (interrupts unmasked)
SR_{5-4}	00 (16-bit offset width)
SR_3	0 (local stack selected)
SR_2	0 (all interrupts disabled)
SR_1	0 (positive sign bit)
SR_0	0 (interrupts latched)

Table 5.2 Status After Reset Operation

Instruction Set 6

6.1 INTRODUCTION

This chapter describes all of the ADSP-1401 instructions. Section 6.2 describes each instruction in detail and provides one or more examples that use the instruction. Section 6.3 summarizes the standard instruction mnemonics and opcodes and gives a brief description of each instruction. Both sections are grouped by the following functions; within each group, the instructions are listed alphabetically by mnemonic.

- Conditional Jump and Branch
- Interrupt Control
- Subroutine Stack
- Register Stack
- Counters
- Status Register
- Relative Jump Offset Width
- Miscellaneous

The following abbreviations are used in this chapter:

Conditions:

CONDITION	A condition represented by a 2-bit code provided in the opcode
SIGN	The sign bit of a specified counter or the SR_1 status register bit
FLAG	The input at the FLAG pin

Subroutine Stack:

SS	Subroutine Stack
SSP	Subroutine Stack Pointer

Register Stack:

GSP	Global Stack Pointer
LSP	Local Stack Pointer
RS	Register Stack (either Global Stack or Local Stack)
RSP	Register Stack Pointer (either GSP or LSP)

6 Instruction Set

Interrupts:

IR_i	*ith* interrupt signal
IRIP	Interrupt In Progress signal
IV	Interrupt Vector
IVP	Interrupt Vector Pointer

Registers and Counters:

SR	Status Register
PC	Program Counter
SLR	Stack Limit Register
R_i	*ith* register on Register Stack (Global Stack or Local Stack)
C_i	*ith* counter

Ports:

D	Data port
Y	Address port

Note that these abbreviations are also used in the mnemonics; PSDSS, for example, is an instruction that pushes the data port value on the subroutine stack.

6.2 INSTRUCTION REFERENCE

This section describes the operations performed by each ADSP-1401 instruction. Any values that are changed and any restrictions that apply are specified. The opcode for each instruction is listed, plus one or more short examples that illustrate usage.

In normal operation, the ADSP-1401 outputs the value of the program counter on the address port during clock HI and increments the program counter at the end of the cycle (the program counter is post-increment). This action is represented by Y $\leftarrow$ PC+1; the update of the program counter is implied, unless it is specifically inhibited by the instruction. Any instruction that outputs an address other than the program counter value automatically loads the program counter with this new address.

In the examples, the code is listed in four columns. The first column contains microcode addresses, if any are used. The second column contains opcodes, and the third column contains the corresponding mnemonics. The fourth column contains data port values, if any are needed.

6.2.1 Conditional Jump and Branch Instructions

Jump and branch instructions control program flow. Jump address sources include the register stack, the data port, and the indirect address space. Jumps can be unconditional or conditioned on the Flag input or counter Sign bit. Two bits of the opcode, *cc*, determine the condition as follows:

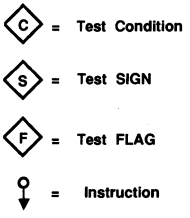
cc	*Condition*
00	Unconditional
01	Not Flag
10	Flag
11	Sign

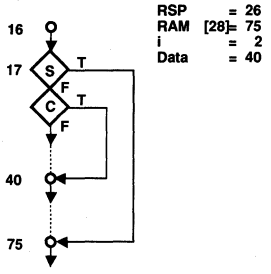
In the opcodes of instructions that affect a counter, a 2-bit code, *ii*, specifies the counter index (3-0). These instructions use the counter's sign bit, rather than the value of SR_1, to test the Sign condition. Instructions that access values from the register stack use the same 2-bit code to specify the register index. If an instruction requires both a counter and a register, the counter and register have the same index.

Flow charts for all conditional jump instructions are shown in Figure 6.1.

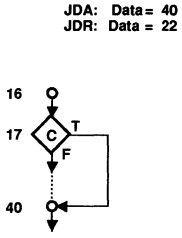
BRANCH	If Sign Jump Register Else If Condition Jump Data, Absolute (Condition ≠ Sign)
JDA	If Condition Jump Data, Absolute
JDI	If Condition Jump Data, Indirect
JDR	If Condition Jump Data, Relative
JDRST	If Sign Jump Data, Absolute, and Reset Counter
JPCNF	If Not Flag Jump PC
JPCOF	If Flag Jump PC
JRC	If Condition Jump Register (Condition ≠ Sign)
JRS	If Sign Jump Register and Decrement Counter
JSA	If Condition Jump Subroutine, Absolute
JSR	If Condition Jump Subroutine, Relative
JTWO	If Condition Jump PC + 2
RTN	If Condition Return From Subroutine

6 Instruction Set

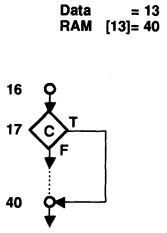




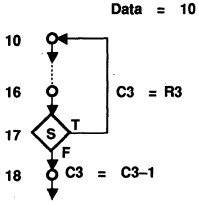
BRANCH



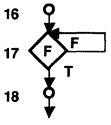
JDA, JDR



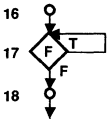
JDI



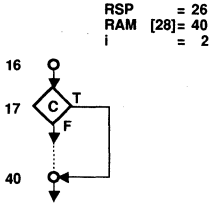
JDRST



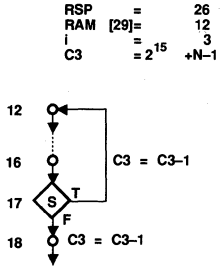
JPCNF



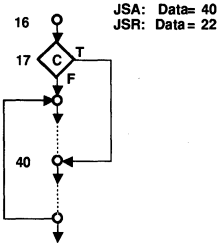
JPCOF



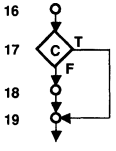
JRC



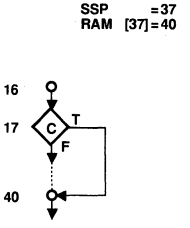
JRS



JSA, JSR



JTWO



RTN

Figure 6.1 Jump Instruction Flow Charts

6.2.1.1 BRANCH

If Counter Sign Jump Register Stack, Decrement Counter
Else If Condition Jump Data Absolute, Decrement Counter
(Condition ≠ Sign)
Else Decrement Counter

IF SIGN OF C_i THEN Y ← R_i , C_i ← C_i – 1

ELSE IF CONDITION THEN Y ← D, C_i ← C_i – 1

ELSE Y ← PC + 1, C_i ← C_i – 1

Description:
Implements a three-way conditional branch. If the sign bit of the specified counter is HI, the next address is the value at the RAM location given by the selected pointer (GSP or LSP) plus *i*, the counter index. If the sign bit of the counter is LO and the specified condition is true, the next address is the value at the data port. If the sign bit of the counter is LO and the specified condition is false, the incremented program counter provides the next address. In all cases, the specified counter is decremented. The Sign condition cannot be specified in the second IF clause because Sign is used explicitly in the first IF clause.

Opcode:

```
100 cc ii
```

Example:

```
4000    000 01 11       SGSP
4001    100 01 11       BRANCH      H#1000
```

SGSP selects the global stack. If the sign bit of C_3 is HI, the address output by BRANCH is [GSP+3]. If the sign bit of C_3 is LO and the Flag input is LO, the address is H#1000. If the sign bit of C_3 is LO and the Flag input is HI, the address is H#4002. In all three cases, C_3 is decremented.

Sign of C_3	*Flag*	*Next Address*
1	x	GSP + i
0	0	Data port value
0	1	PC + 1

6.2.1.2 JDA

If Condition Jump Data Absolute

IF CONDITION
THEN Y ← D
ELSE Y ← PC + 1

Description:
Implements a conditional jump to a specified address. If the specified condition is true, the next address is read from the data port. Otherwise, the incremented program counter provides the next address.

Opcode:
```
111 cc 11
```

Example:
```
1000    011 00 00      DCCNTR
1001    111 11 11      JDA          H#11FF
```

DCCNTR transfers the sign bit of C_0 to SR_1 and then decrements C_0. If the resulting SR_1 is HI, JDA executes a jump to H#11FF.

6.2.1.3 JDI

If Condition Jump Data Indirect

IF CONDITION
THEN Y ← $[D_{5\text{-}0}]$
ELSE Y ← PC + 1

Description:
Implements a conditional indirect jump. If the specified condition is true, the six least significant bits of the data port ($D_{5\text{-}0}$) address a location in internal RAM; the value stored at this location is the next address. Otherwise, the incremented program counter provides the next address.

Opcode:

```
101 cc 10
```

Example:

```
1000    101 00 10       JDI          H#0030
```

The value of [H#30] is output on the address port unconditionally.

6.2.1.4 JDR

If Condition Jump Data Relative

IF CONDITION
THEN Y ← PC + D + 1
ELSE Y ← PC + 1

Description:
Implements a conditional jump to an address relative to the current address. If the specified condition is true, the next address is the (twos-complement) value on the data port added to the incremented program counter value. Otherwise, the incremented program counter provides the next address. The width of the data port input (16 bits, 12 bits, or 8 bits) is determined by SR_{5-4}.

Opcode:

```
111 cc 01
```

Examples:

```
1000    111 10 01       JDR          H#0004
```

If Flag is HI, address H#1005 is output and the next four instructions (H#1001 to H#1004) are skipped.

```
9FFF    111 10 01       JDR          H#8000
```

If Flag is HI, address H#2000 (H#A000 + H#8000) is output.

6.2.1.5 JDRST

If Counter Sign Jump Data Absolute and Reinitialize Counter
Else Decrement Counter

IF SIGN OF C_i
THEN Y ← D, C_i ← R_i
ELSE Y ← PC + 1, C_i ← C_i – 1

Description:
Implements either a jump and counter reinitialization or a counter decrement. If the sign bit of the specified counter (not SR_1) is HI, the next address is read from the data port, and the counter is reinitialized with the value from the register that is *i* (the counter index) locations greater than the selected register stack pointer (GSP or LSP). Otherwise, the incremented program counter provides the next address, and the specified counter is decremented.

Opcode:

```
100 11 ii
```

Example:

```
0FEE    000 01 10      SLSP
0FEF    011 10 01      WRCNTR      H#0001
0FF0    100 11 01      JDRST       H#1000
                |
          (instructions)
                |
0FFF    111 00 11      JDA         H#0FF0
1000    000 00 00      CONT
```

The LSP is selected and the C_1 counter is initialized to H#0001 before the loop (instructions H#0FF0 to H#0FFF) is entered. The first time JDRST is executed, the C_1 sign bit is LO, so C_1 is decremented and the loop is executed. JDA performs an unconditional jump to the top of the loop. The second time JDRST is executed, C_1 is H#0000, so the loop is executed again. The third time JDRST is executed, C_1 is H#FFFF; the C_1 sign bit is HI, so the loop is exited by a jump to H#1000, and C_1 is loaded with the value [LSP+1].

6.2.1.6 JPCNF

If Not Flag Repeat Instruction

IF NOT FLAG
THEN Y ← PC
ELSE Y ← PC + 1

Description:
Repeats execution of the current instruction under control of the Flag signal. The Flag input must meet setup requirements similar to those of the instruction port. If the Flag input is inactive (LO), the program counter increment is inhibited; otherwise, the program counter is incremented.

Opcode:

```
011 0101
```

Example:

```
1000    011 0101        JPCNF
```

If Flag is LO, address H#1000 is output again, and the microcode instruction at that address is repeated.

6.2.1.7 JPCOF

If Flag Repeat Instruction

IF FLAG
THEN Y ← PC
ELSE Y ← PC + 1

Description:
Repeats execution of the current instruction under control of the Flag signal. The Flag input must meet setup requirements similar to those of the instruction port. If the Flag input is active (HI), the program counter increment is inhibited; otherwise, the program counter is incremented.

Opcode:
```
001 0101
```

Example:
```
1000    001 0101        JPCOF
```

If Flag is HI, address H#1000 is output again, and the microcode instruction at that address is repeated.

6.2.1.8 JRC

If Condition Jump Register Stack (Condition ≠ Sign)

IF CONDITION
THEN Y ← R_i
ELSE Y ← PC + 1

Description:
Implements a conditional jump to an address stored on the register stack. If the specified condition is true, the next address is the value of the register that is *i* (the register stack index) locations above the selected pointer (GSP or LSP). Otherwise, the incremented program counter provides the next address.

The Sign condition cannot specified for this instruction. However, the JRS instruction performs the equivalent operation for the Sign condition and decrements a specified counter as well.

Opcode:

```
110 cc ii
```

Example:

```
1000    000 01 11      SGSP
1001    000 11 00      WRRSP       H#0020
1002    110 00 11      JRC
```

The GSP is selected and loaded with the value H#20. JRC performs a jump to [H#23] unconditionally.

6.2.1.9 JRS

If Counter Sign Jump Register Stack and Decrement Counter
Else Decrement Counter

IF SIGN OF C_i
THEN $Y \leftarrow R_i$, $C_i \leftarrow C_i - 1$
ELSE $Y \leftarrow PC + 1$, $C_i \leftarrow C_i - 1$

Description:
Implements a conditional jump to an address stored in the register stack, and decrements the specified counter. If the sign of the specified counter is HI, the next address is the value of the register that is *i* (the register stack index) locations above the selected pointer (GSP or LSP). Otherwise, the incremented program counter provides the next address. The specified counter is decremented in either case.

Opcode:
```
110 11 ii
```

Example:
```
1000    000 01 10      SLSP
1001    000 11 00      WRRSP       H#0030
1002    110 11 00      JRS
```

The LSP is selected and loaded with the value H#30. If the sign bit of C_0 is HI, a jump to [H#30] is executed. C_0 is decremented whether or not the jump is executed.

6.2.1.10 JSA

If Condition Jump Subroutine, Data Absolute

IF CONDITION
THEN Y ← D, SS ← PC + 1
ELSE Y ← PC + 1

Description:
Implements a conditional jump to a subroutine that begins at the specified address. If the specified condition is true, the next address is the data port value, and the incremented program counter value is pushed on the subroutine stack as the return address. Otherwise, the incremented program counter provides the next address.

Opcode:

```
111 cc 00
```

Example:

```
1000    110 11 10      JRS
1001    111 11 00      JSA          H#2000
```

The JRS instruction decrements C_2, transferring its sign bit to SR_1 before the decrement. If this sign bit is HI, JSA jumps to the subroutine at H#2000.

6.2.1.11 JSR

If Condition Jump Subroutine, Data Relative

IF CONDITION
THEN Y ← PC + 1 + D, SS ← PC + 1
ELSE Y ← PC + 1

Description:
Implements a conditional jump to a subroutine that begins at an address relative to the current address. If the specified condition is true, the next address is the (twos-complement) value on the data port added to the incremented program counter value, and the incremented program counter value is pushed on the subroutine stack as the return address. Otherwise, the incremented program counter provides the next address. The width of the data port input (16 bits, 12 bits, or 8 bits) is determined by SR_{5-4}.

Opcode:
```
111 cc 10
```

Example:
```
1000    111 10 10      JSR          H#0020
```

If Flag is HI, the address H#1021 is output; otherwise, the address H#1001 is output.

6.2.1.12 JTWO

If Condition Skip One Address

IF CONDITION
THEN Y ← PC + 2
ELSE Y ← PC + 1

Description:
Conditionally skips one instruction without requiring a jump address. If the specified condition is true, the next address is the incremented program counter value incremented a second time. Otherwise, the normal (incremented once) program counter value is the next address.

Opcode:

```
101 cc 01
```

Examples:

```
1000    101 01 01      JTWO
1001    111 00 11      JDA
1002    000 00 00      CONT
```

If Flag is LO, the address H#1002 is output, and the unconditional jump performed by the JDA instruction is skipped.

```
1000    101 10 01      JTWO
1001    101 01 01      JTWO
1002    101 10 01      JTWO
1003    101 01 01      JTWO
1004    000 00 00      CONT
1005    000 00 00      CONT
```

If Flag is HI, instructions at H#1000, H#1002 and H#1004 are executed. If Flag is LO, instructions at H#1001, H#1003 and H#1005 are executed.

6.2.1.13 RTN

If Condition Return From Subroutine

IF CONDITION
THEN Y ← SS
ELSE Y ← PC + 1

Description:
Implements a conditional return from a subroutine (jump to the return address). If the specified condition is true, the return address is popped from the subroutine stack and output as the next address. Otherwise, the incremented program counter provides the next address.

Opcode:
```
101 cc 11
```

Example:
```
1000    101 10 11      RTN
```

If Flag is HI, the return address is popped from the subroutine stack and output on the address port.

6.2.2 Interrupt Control Instructions

Interrupt control instructions affect interrupt handling operations. Some instructions set and clear bits of the interrupt mask, enable or disable interrupt handling, and select latched or transparent interrupts. Some instructions allow you to read and write the interrupt vector file. Other instructions, executed within an interrupt service routine, control the completion of the service routine and the nesting of subsequent service routines. Note that the SLRIVP, a dual-purpose instruction, affects the Stack Limit Register (SLR) as well as the Interrupt Vector Pointer (IVP).

CAIR	Clear All Interrupts
CCIR	Clear Current Interrupt
DISIR	Disable All Interrupts
ENAIR	Enable All Interrupts
IRMBC	Interrupt Mask Bitwise Clear
IRMBS	Interrupt Mask Bitwise Set
RDIV	Read Interrupt Vector At Data Port
RTNIR	Return From Interrupt Routine
SLIR	Select Latched Interrupts
SLRIVP	Write SLR and IVP From Data Port
STIR	Select Transparent Interrupts
WRIV	Write Interrupt Vector From Data Port

6.2.2.1 CAIR

Clear All External Interrupts

$IR_i \leftarrow 0$

IRIP $\leftarrow 0$

Y $\leftarrow$ PC + 1

Description:
Clears all external interrupt latches (IR_{8-1}) and the IRIP (Interrupt In Progress) signal. This instruction does *not* clear internal interrupts IR_9 and IR_0; these are cleared by the SLRIVP and CLRS instructions, respectively. Executing this instruction from within a service routine allows the nesting of a subsequent service routine by re-enabling interrupt processing, which is normally inhibited by IRIP. The effect of re-enabling interrupt processing is to convert the service routine to a subroutine.

The RTNIR instruction also clears IRIP and the latch for the interrupt being serviced (external interrupts only); however, unlike RTNIR, CAIR clears all external interrupt latches and does not pop the return address.

Opcode:
```
000 0001
```

Example:
```
;Interrupt service routine
1000   000 0000      CONT
1001   000 0001      CAIR
1002   000 0000      CONT
```

The first instruction of an interrupt service routine must always be a CONT. The CAIR instruction clears pending external interrupts and IRIP. If an IR_9 or IR_0 interrupt is pending when CAIR is executed, its service routine vector is output in the cycle following the CAIR instruction (during execution of the CONT instruction at H#1002). The IR_9 or IR_0 service routine stores H#1003 as its return address.

6.2.2.2 CCIR

Clear Current (External) Interrupt

$IR_{current} \leftarrow 0$

$IRIP \leftarrow 0$

$Y \leftarrow PC + 1$

Description:

Clears the latch of the external interrupt currently being serviced and enables the processing of the next interrupt by clearing the IRIP (Interrupt In Progress) signal. Internal interrupts (IR_9 and IR_0) are *not* cleared; IR_9 and IR_0 are cleared by the SLRIVP and CLRS instructions, respectively. This instruction can enable interrupts during the execution of a service routine, allowing the nesting of one service routine within another, which is normally inhibited by IRIP. The effect of re-enabling interrupt processing is to convert the service routine to a subroutine.

The RTNIR instruction also clears IRIP and the latch for the interrupt being serviced (external interrupts only); however, unlike RTNIR, CCIR does not pop the return address.

Opcode:

```
001 0001
```

Example:

```
;Interrupt service routine
1000    000 0000      CONT
1001    001 0001      CCIR
1002    000 0000      CONT
```

The first instruction of the service routine must be the CONT instruction, to allow for the push of the return address push on the subroutine stack. The CCIR instruction clears IRIP and the interrupt latch (for the current interrupt only). If another interrupt is pending, its vector is output in the following cycle (the CONT instruction at H#1002), and its service routine begins on the cycle after that. Address H#1003 is stored as the return address.

6.2.2.3 DISIR

Disable All Interrupts

$SR_2 \leftarrow 0$

$Y \leftarrow PC + 1$

Description:

Disables all interrupt processing by clearing bit 2 of the status register. This instruction takes effect in the cycle following its execution; interrupts that occur while DISIR is being executed are still processed. When interrupts are disabled, no interrupts are processed regardless of the values in the interrupt mask (SR_{15-6}). If interrupt latching is enabled(SR_0 is LO), external interrupts (IR_{8-1}) continue to be latched while they are disabled. Note that the nonmaskable Trap interrupt is still enabled.

Opcode:

```
001 0110
```

Example:

```
1000    001 0110        DISIR
            |
        (instructions)
            |
1010    011 0110        ENAIR
```

The DISIR instruction is used here to guarantee that instructions are executed without interruption. An interrupt vector can be output in the cycle after DISIR (H#1001) if an interrupt occurs while DISIR is executing. Interrupts are recognized in the cycle that ENAIR executes and a vector can be output in the following cycle. The protected microcode is thus H#1002 to H#1011, inclusive.

6.2.2.4 ENAIR

Enable All Interrupts

$SR_2 \leftarrow 1$

$Y \leftarrow PC + 1$

Description:
Enables all interrupt processing by setting bit 2 of the status register. The interrupt mask (SR_{15-6}) enables and disables individual interrupts. If an interrupt is pending when ENAIR is executed, the interrupt vector is output on the next cycle.

Opcode:

```
011 0110
```

Example:

```
1000    001 0110        DISIR
            |
        (instructions)
            |
1010    011 0110        ENAIR
1011    000 0000        CONT
```

Interrupts that occur while interrupts are disabled are latched. When interrupts are re-enabled, the vector for a pending interrupt is output on the cycle following the ENAIR instruction (in the CONT instruction). The service routine begins on the cycle after that, and H#1012 is stored as the return address.

6.2.2.5 IRMBC

Interrupt Mask Bitwise Clear

$SR_{15-6} \leftarrow NOT\ (D_{15-6})\ AND\ SR_{15-6}$
$Y \leftarrow PC + 1$

Description:

Clears each of the interrupt mask bits (SR_{15-6}) for which the corresponding data port bit (D_{15-6}) is set. D_{5-0} are ignored.

Opcode:

```
001 0011
```

Example:

```
1000    001 0011        IRBMC       H#0C00
```

This instruction clears the interrupt mask bits for IR_{5-4} as follows:

Status register (IR_{5-0} masked)	0000 1111 11xx xxxx
H#0C00	0000 1100 0000 0000
Status register (IR_{3-0} masked)	0000 0011 11xx xxxx

6.2.2.6 IRMBS

Interrupt Mask Bitwise Set

$SR_{15-6} \leftarrow D_{15-6}$ OR SR_{15-6}
$Y \leftarrow PC + 1$

Description:

Sets each of the interrupt mask bits (SR_{15-6}) for which the corresponding data port bit (D_{15-6}) is set. D_{5-0} are ignored.

Opcode:

```
001 0010
```

Example:

```
1000    001 0010        IRBMS       H#3000
```

This instruction sets the interrupt mask bits for IR_{7-6} as follows:

Status register (IR_{5-0} masked)	0000 1111 11xx xxxx
H#3000	0011 0000 0000 0000
Status register (IR_{7-0} masked)	0011 1111 11xx xxxx

6.2.2.7 RDIV

Read Interrupt Vector File at Data Port

D ← IV

IVP ← IVP + 1

Y ← PC + 1

Description:
Outputs the value of the interrupt vector specified by the IVP on the data port and increments the IVP. You should disable interrupts before executing the RDIV instruction.

Opcode:
```
010 1101
```

Example:
```
1000    001 0110        DISIR
1001    001 1101        SLRIVP      H#0020
1002    010 1101        RDIV
1003    010 1101        RDIV
1004    010 1101        RDIV
1005    011 0110        ENAIR
```

The SLRIVP instruction loads both the IVP and the SLR; in this case, the IVP points to location 0. Because the RDIV instruction automatically increments the IVP, three sequential executions of the RDIV instruction output the interrupt vectors for IR_0, IR_1, and IR_2 on the data port.

6.2.2.8 RTNIR

Return From Interrupt Service Routine

$IR_{current} \leftarrow 0$

IRIP $\leftarrow 0$

Y $\leftarrow$ SS

Description:

Executes a return from an interrupt service routine (jump to return address) and re-enables the processing of interrupts. This instruction clears the latch of the current external interrupt (IR_{8-1}), clears the IRIP (Interrupt In Progress) signal, and pops the return address from the subroutine stack. If an interrupt is pending when the RTNIR instruction is executed, it is recognized on the following cycle, and the first instruction of its service routine is executed on the cycle after that.

Internal interrupt latches (IR_9 and IR_0) are not affected and must be cleared explicitly (using SLRIVP or CLRS, respectively) before the return is executed. Note that whereas a subroutine return (RTN) can be conditioned, a service routine return (RTNIR) is always unconditional.

Opcode:

```
000 0011
```

Example:

```
1000    001 0100        CLRS
1001    000 0011        RTNIR
```

These two instructions typically end an IR_0 service routine. The CLRS instruction clears the IR_0 interrupt latch. The RTNIR instruction clears the IRIP signal (if it has not been already cleared by a CAIR or CCIR instruction) and pops the return address from the subroutine stack.

6.2.2.9 SLIR

Select Latched Interrupts

$SR_0 \leftarrow 0$

$Y \leftarrow PC + 1$

Description:
Enables the latching of external interrupts (IR_{8-1}). IR_{8-5} are latched during clock HI and IR_{4-1} are latched during clock LO. When latching is enabled, all interrupts are latched even if particular interrupts are masked or all interrupts are disabled.

Opcode:
```
001 0111
```

Example:
```
1000    011 0111        STIR
            |
        (instructions)
            |
1010    001 0111        SLIR
```

In this example, interrupt latching is disabled for microcode instructions from H#1001 to H#1010. The SLIR instruction re-enables interrupt latches.

6.2.2.10 SLRIVP

Transfer From Data Port to SLR and IVP

$SLR \leftarrow D_{5-2}$

$IVP \leftarrow D_{15-12}$

$IR_9 \leftarrow 0$

$Y \leftarrow PC + 1$

Description:
Loads the SLR from D_{5-2}, loads the IVP from D_{15-12} and clears the IR_9 interrupt. D_{11-6} and D_{1-0} are ignored. The SLR determines the boundary between the subroutine stack and register stack. The 4-bit SLR stores the four most significant bits of the subroutine stack limit. The remaining two bits are zero-filled; therefore, the subroutine stack limit can be set only at addresses that are multiples of four. The IVP points to a location in the interrupt vector file and is used to load and dump vector values. Neither the SLR nor the IVP can be read at the data port. The IR_9 interrupt can be cleared only through this instruction.

Opcode:

```
001 1101
```

Example:

```
1000    001 1101        SLRIVP      H#5024
```

This instruction sets the IVP to location 5 and the SLR to H#24 and clears IR_9 as well. D_{11-6} and D_{1-0} are all zeros in this example, but could be set to any values.

6.2.2.11 STIR

Select Transparent Interrupts

$SR_0 \leftarrow 1$

$IR_{8-1} \leftarrow 0$

$Y \leftarrow PC + 1$

Description:
Disables the latching of external interrupts (IR_{8-1}) by setting SR_0. In transparent interrupt mode, an external interrupt signal must remain active until it is recognized by the ADSP-1401. IR_0 and IR_9 are still latched and must be cleared using CLRS and SLRIVP, respectively. The STIR instruction clears the interrupt latch for all pending external interrupts (IR_{8-1}).

Opcode:

```
011 0111
```

Example:

```
1000    011 0111        STIR
1001    001 0110        DISIR
            |
        (instructions)
            |
1010    011 0110        ENAIR
1011    001 0111        SLIR
```

The STIR instruction disables the interrupt latch. Interrupts that occur while interrupt processing is disabled (from H#1002 to H#1010) are ignored. No interrupts will be pending when the ENAIR instruction re-enables interrupt processing.

6.2.2.12 WRIV

Transfer From Data Port to Interrupt Vector File

IV ← D

IVP ← IVP + 1

Y ← PC + 1

Description:
Loads the interrupt vector specified by the IVP from the data port and increments the IVP. Use this instruction to preload the register file or to replace an interrupt vector with a new vector. You should disable all interrupts before executing the WRIV instruction.

Opcode:
```
000 1101
```

Example:
```
1000    001 0110        DISIR
1001    001 1101        SLRIVP      H#7020
1002    000 1101        WRIV        H#FE00
1003    011 0110        ENAIR
```

The DISIR instruction disables interrupts. The SLRIVP instruction loads the SLR and the IVP; in this case, the IVP is loaded with H#7. The WRIV instruction loads the interrupt vector 7 with the value H#FE00 and increments the IVP. The ENAIR instruction re-enables interrupts.

6.2.3 Subroutine Stack Instructions

Subroutine stack instructions push and pop the subroutine stack or manipulate the Subroutine Stack Pointer (SSP).

DSSP	Decrement SSP By One
PPSSD	Pop Subroutine Stack To Data Port
PSDSS	Push Data Port To Subroutine Stack
RDSSP	Read SSP At Data Port
WRSSP	Write SSP From Data Port

6.2.3.1 DSSP

Decrement SSP

SSP ← SSP – 1

Y ← PC + 1

Description:
Decrements the SSP. This instruction removes one location from the top of the subroutine stack (the data at that location is unchanged).

Opcode:

```
000 0010
```

Example:

```
1000    000 0101        PSGSP
1001    010 0001        PSSR
                |
        (instructions)
                |
1010    000 0010        DSSP
1011    000 0010        DSSP
```

The first two instructions push the GSP and the status register on the subroutine stack. If the intervening instructions do not affect the subroutine stack, the two DSSP instructions remove the status register and GSP values (in that order) from the stack without affecting the current status register and GSP values.

6.2.3.2 PPSSD

Pop Subroutine Stack to Data Port

D ← SS

Y ← PC + 1

Description:
Outputs the value at the stack location specified by the SSP on the data port and decrements the SSP.

Opcode:
```
011 1110
```

Example:
```
1000    000 1011        PSCNTR
1001    011 1110        PPSSD
```

Counters cannot be read directly. These instructions allow C_3 to be read at data port. The PSCNTR instruction pushes the value of counter C_3 on the subroutine stack. The PPSSD instruction pops the C_3 value to data port.

6.2.3.3 PSDSS

Push Data Port to Subroutine Stack

SS ← D

Y ← PC + 1

Description:
Increments the SSP and loads the stack location specified by the SSP with the data port value.

Opcode:
001 1110

Example:

```
;Subroutine
2000    000 0010        DSSP
2001    001 1110        PSDSS       H#1000
```

When the subroutine is entered, the return address is on the top of the subroutine stack. The DSSP instruction decrements the SSP, and the PSDSS instruction pushes H#1000 onto the stack, overwriting the old return address.

6.2.3.4 RDSSP

Transfer From SSP to Data Port

$D_{5-0} \leftarrow SSP$

$Y \leftarrow PC + 1$

Description:
Outputs the value of the SSP on six least significant data port bits (D_{5-0}).

Opcode:

```
010 1100
```

Example:

```
1000    010 1100        RDSSP
```

The SSP value is output on D_{5-0}. D_{15-6} are undefined and should be masked out.

6.2.3.5 WRSSP

Transfer From Data Port to SSP

$SSP \leftarrow D_{5-0}$

$Y \leftarrow PC + 1$

Description:
Loads the SSP from the six least significant data port bits D_{5-0}.

Opcode:

```
000 1110
```

Example:

```
1000    000 1110        WRSSP       H#003F
```

The WRSSP instruction initializes the SSP to location H#3F (the last location in the internal RAM), so that the subroutine stack will begin at location 0. IR_9 must be disabled when this location is used, as explained in the application note *Stack Paging Expands Internal RAM of the ADSP-1401 Program Sequencer.*

6.2.4 Register Stack Instructions

Register stack instructions affect either the global stack or local stack, depending on the value of a status register bit (SR_3). Two instructions select a stack by controlling the value of this bit. Other instructions push or pop the selected stack, or manipulate the selected stack pointer (GSP or LSP). The GSP can be pushed to or popped from the subroutine stack.

AIRSP	Add *i* To RSP
PPGSP	Pop Subroutine Stack To GSP
PPRSD	Pop Register Stack To Data Port
PSDRS	Push Data Port To Register Stack
PSGSP	Push GSP To Subroutine Stack
PSPC	Push Program Counter To Register Stack
RDRSP	Read RSP At Data Port
WRRSP	Write RSP From Data Port
S1RSP	Subtract 1 From RSP
S4RSP	Subtract 4 From RSP
SGSP	Select GSP
SLSP	Select LSP

6.2.4.1 AIRSP

Add *i* to RSP

RSP ← RSP + i

Y ← PC + 1

Description:
Adds the specified *i* value to the selected RSP (GSP or LSP). The two-bit *i* values 00, 01, 10, and 11 represent the numbers 4, 1, 2, and 3, respectively. This instruction, by increasing the RSP, effectively removes one to four registers from the register stack (although the stack data is unchanged).

Opcode:

```
010 10 ii
```

Examples:

```
1000    000 01 10       SLSP
1001    010 10 10       AIRSP
```

The SLSP instruction selects the LSP, and the AIRSP instruction adds two to the LSP, removing two registers from the top of the local stack.

```
1000    000 01 11       SGSP
1001    010 10 00       AIRSP
```

The SGSP instruction selects the GSP, and the AIRSP instruction adds four to the GSP, removing four registers from the top of the global stack.

6.2.4.2 PPGSP

Pop Subroutine Stack to GSP

GSP ← SS

Y ← PC + 1

Description:
Loads the GSP with the value at the stack location specified by the SSP and decrements the SSP.

Opcode:
```
000 0100
```

Example:
```
1000    000 0101        PSGSP
1001    000 0111        SGSP
1002    000 1100        WRRSP       H#0026
            |
        (instructions)
            |
1010    000 0100        PPGSP
```

In this example, the GSP is saved on the subroutine stack before it is overwritten, then restored later. The PSGSP instruction pushes the GSP on the subroutine stack. The GSP is then selected and written with a new value (H#26). Several instructions later, assuming that the subroutine stack has not been altered, the old GSP value is restored by the PPGSP instruction.

6.2.4.3 PPRSD

Pop Register Stack to Data Port

D ← RS

Y ← PC + 1

Description:
Outputs the value of the location specified by the selected RSP (GSP or LSP) on the data port and increments the pointer.

Opcode:
```
011 1111
```

Example:
```
1000    000 0111        SGSP
1001    011 1111        PPRSD
1002    011 1111        PPRSD
```

The GSP is selected, then PPRSD is executed twice to pop the top two global registers to the data port.

6.2.4.4 PSDRS

Push From Data Port to Register Stack

RS ← D

Y ← PC + 1

Description:
Decrements the selected RSP (GSP or LSP) and loads the stack location specified by the pointer with the data port value.

Opcode:
```
001 1111
```

Example:
```
1000    000 0111        SGSP
1001    001 1111        PSDRS       H#1000
1002    001 1111        PSDRS       H#2000
1003    001 1111        PSDRS       H#3000
```

In this example, global jump addresses are loaded from the data port. The GSP is selected, then the global stack is pushed with three values from data port using the PSDRS instruction. Indirect jump addresses can be loaded in this way as well.

6.2.4.5 PSGSP

Push GSP to Subroutine Stack

SS ← GSP

Y ← PC + 1

Description:
Increments the SSP and loads the stack location specified by the SSP with the value of the GSP.

Opcode:
```
000 0101
```

Example:
```
1000    000 0101        PSGSP
1001    000 0111        SGSP
1002    000 1100        WRRSP       H#0026
            |
        (instructions)
            |
100F    000 0100        PPGSP
```

In this example, the GSP is saved on the subroutine stack before it is overwritten, then restored later. The PSGSP instruction pushes the GSP on the subroutine stack. The GSP is then selected and written with a new value (H#26). Several instructions later, assuming that the subroutine stack has not been altered, the old GSP value is restored by the PPGSP instruction.

6.2.4.6 PSPC

Push Program Counter to Register Stack

RS ← PC + 1

Y ← PC + 1

Description:
Decrements the selected RSP (GSP or LSP) and loads the stack location specified by the pointer with the incremented program counter value.

Opcode:
```
010 0011
```

Example:
```
1000    000 0110        SLSP
1001    010 0011        PSPC
           |
        (instructions)
           |
100F    110 0100        JRC
```

This example uses the PSPC instruction to push the top-of-loop address on the local stack. The local stack is selected by the SLSP instruction. Then the PSPC instruction pushes the incremented program counter value (H#1002) on the local stack. At the bottom of the loop, if Flag is LO, the JRC instruction performs a jump to [LSP+0] = H#1002.

6.2.4.7 RDRSP

Transfer RSP to Data Port

$D_{5\text{-}0} \leftarrow RSP$

$Y \leftarrow PC + 1$

Description:
Outputs the value of the selected RSP (GSP or LSP) on six least significant data bits, $D_{5\text{-}0}$. $D_{15\text{-}6}$ are undefined and should be masked.

Opcode:
```
010 1111
```

Example:
```
1000    000 0110        SLSP
1001    010 1111        RDRSP
1002    000 0111        SGSP
1003    010 1111        RDRSP
```

These instructions allow the register stack pointers to be read at the data port. The first two instructions select the LSP and transfer its value to $D_{5\text{-}0}$. The last two instructions select the GSP and transfer its value to $D_{5\text{-}0}$.

6.2.4.8 S1RSP

Subtract 1 From RSP

RSP ← RSP – 1

Y ← PC + 1

Description:
Subtracts one from the specified RSP (GSP or LSP). This instruction can be used to recover the most recently popped register.

Opcode:
```
000 1111
```

Example:
```
1000    000 0110        SLSP
1001    011 1111        PPRSD
            |
        (instructions)
            |
100F    000 1111        S1RSP
1010    011 1111        PPRSD
```

Because popping the stack does not affect stack data, decreasing the value of the GSP or LSP allows access to registers that have been previously popped. The first two instructions pop the top local stack register to the data port. Later, assuming the intervening instructions do not alter the local stack, the S1RSP instruction recovers the register so it can be popped again.

6.2.4.9 S4RSP

Subtract 4 From RSP

RSP ← RSP – 4

Y ← PC + 1

Description:
Subtracts four from the specified RSP (GSP or LSP). This instruction can be used to recover the four most recently used registers.

Opcode:
```
011 1100
```

Example:
```
1000    111 0000        JSA         H#1FFF
1001    000 0110        SLSP
1002    011 1100        S4RSP
```

The JSA instruction jumps to a subroutine at address H#1FFF. The subroutine (not shown) pushes four jump addresses on the local stack; it pops these registers before returning to the main program. After the return from the subroutine, the LSP is selected and the S4RSP instruction recovers the four jump addresses.

6.2.4.10 SGSP

Select GSP

$SR_3 \leftarrow 1$

$Y \leftarrow PC + 1$

Description:
Selects the GSP by setting bit 3 in the status register. All register stack instructions executed while the GSP is selected affect the global stack.

Opcode:

```
000 0111
```

Example:

```
1000    000 0111        SGSP
1001    010 0011        PSPC
```

The SGSP instruction selects the GSP so that the PSPC instruction pushes the incremented program counter (H#1002) on the global stack.

6.2.4.11 SLSP

Select LSP

$SR_3 \leftarrow 0$

$Y \leftarrow PC + 1$

Description:
Selects the LSP by clearing bit 3 in the status register. All register stack instructions executed while the LSP is selected affect the local stack.

Opcode:

```
000 0110
```

Example:

```
1000    000 0110        SLSP
1001    110 0011        JRC
```

The SLSP instruction selects the LSP so that the JRC instruction performs a jump to [LSP+3].

6.2.4.12 WRRSP

Transfer From Data Port to RSP

$RSP \leftarrow D_{5-0}$

$Y \leftarrow PC + 1$

Description:
Loads the selected RSP (GSP or LSP) from the six least significant data bits (D_{5-0}).

Opcode:

```
000 1100
```

Example:

```
1000    000 0111        SGSP
1001    000 1100        WRRSP       H#0030
1002    000 0110        SLSP
1003    000 1100        WRRSP       H#002C
```

This example uses the WRRSP instruction to load the GSP with H#30 and the LSP with H#2C. Although D_{15-6} are zeros in this case, they can be set to any values because they are ignored.

6.2.5 Counter Instructions

Counter instructions affect the contents of a counter or the sign bit in the status register (SR_1). In the opcodes for instructions that affect a specific counter, a 2-bit code, *ii*, specifies the counter index (3-0). One counter instruction (IFCDEC) is conditional; the condition is specified by two bits in the opcode, *cc*, as in conditional jump instructions:

cc	*Condition*
00	Unconditional
01	Not Flag
10	Flag
11	Sign

Counters can be pushed to or popped from the subroutine stack.

CLRS	Clear Sign Bit (SR_1)
DCCNTR	Decrement Counter By 1
IFCDEC	If Condition Decrement C_0
PPCNTR	Pop Subroutine Stack To Counter
PSCNTR	Push Counter To Subroutine Stack
SETS	Set Sign Bit (SR_1)
WRCNTR	Write Counter From Data Port

6.2.5.1 CLRS

Clear Sign Bit and IR_0 Latch

$SR_1 \leftarrow 0$

$IR_0 \leftarrow 0$

$Y \leftarrow PC + 1$

Description:
Clears the sign bit (bit 1 in the status register) and the IR_0 interrupt latch. The sign bit normally contains a copy of the sign bit of the most recently decremented counter (value before decrement). The IR_0 interrupt is activated when the sign bit of a counter goes high. This interrupt is not cleared by the the RTNIR instruction; it must be cleared explicitly using CLRS.

Opcode:

```
001 0100
```

Example:

```
;IR0 service routine
1000    000 0000        CONT
1001    011 1001        WRCNTR      H#0010
1002    001 0100        CLRS
1003    000 0011        RTNIR
```

This IR_0 service routine reinitializes C_1 and returns. When the C_1 counter underflows, its sign bit is written to SR_1, and the IR_0 interrupt is generated. The first instruction of the service routine must be a CONT to allow for the return address push on the subroutine stack. The WRCNTR instruction loads C_1 with a new value from the data port (H#0010). The CLRS instruction clears SR_1 because the counter is no longer underflowed. CLRS must also be executed to clear IR_0 before returning to the main progam; otherwise, the active IR_0 interrupt would cause another execution of the service routine.

6.2.5.2 DCCNTR

Decrement Counter

$C_i \leftarrow C_i - 1$

$Y \leftarrow PC + 1$

Description:
Decrements the specified counter unconditionally.

Opcode:

```
011 00 ii
```

Example:

```
1000    011 0000        DCCNTR
1001    101 1111        RTN
```

The value of the C_0 sign bit is written to SR_1 before the decrement. The RTN instruction is then conditioned on this value of SR_1.

6.2.5.3 IFCDEC

If Condition Decrement Counter C_0

IF CONDITION
THEN $C_0 \leftarrow C_0 - 1$
$Y \leftarrow PC + 1$

Description:
Decrements counter C_0 if the specified condition is true. The Sign condition tests the value of SR_1, *not* the C_0 sign bit. If IR_0 is enabled to detect counter underflow when this instruction is used to decrement a counter, the interrupt is generated when SR_1 goes HI, *not* when the counter underflows. The effect is a one-decrement lag compared with using IR_0 with other instructions that decrement a counter. One more counter decrement or an initial counter value of one less is needed to generate IR_0.

Opcode:
```
101 cc 00
```

Examples:
```
1000    101 10 00       IFCDEC
```

If Flag is LO, C_0 is unchanged; if Flag is HI, C_0 is decremented.

```
1000    011 00 11       DCCNTR
1001    101 11 00       IFCDEC
```

The sign bit of C_3 is written to SR_1 before the unconditional decrement. The IFCDEC instruction then tests the SR_1 bit to determine whether to decrement C_0.

6.2.5.4 PPCNTR

Pop Subroutine Stack to Counter

$C_i \leftarrow SS$

$Y \leftarrow PC + 1$

Description:
Loads the specified counter with the value at the location specified by the SSP and decrements the SSP.

Opcode:

```
001 10 ii
```

Example:

```
1000    000 10 01       PSCNTR
1001    011 10 01       WRCNTR      H#0011
            |
        (instructions)
            |
1010    001 10 01       PPCNTR
```

This example uses C_1 and later restores its value. The PSCNTR instruction pushes C_1 on the subroutine stack to save its value. The WRCNTR instruction loads a new value (H#0011) to C_1. Later, the old C_1 value is restored by the PPCNTR instruction, assuming that the intervening instructions do not alter the subroutine stack.

6.2.5.5 PSCNTR

Push Counter to Subroutine Stack

$SS \leftarrow C_i$

$Y \leftarrow PC + 1$

Description:
Increments the SSP and loads the stack location specified by the SSP with the value of the specified counter.

Opcode:

```
000 10 ii
```

Example:

```
1000    000 10 10       PSCNTR
1001    011 10 10       WRCNTR      H#0006
            |
        (instructions)
            |
1010    001 10 10       PPCNTR
```

This example uses C_2 and later restores its value. The PSCNTR instruction pushes C_2 on the subroutine stack to save its value. The WRCNTR instruction loads a new value (H#0006) to C_2. Later, the old C_2 value is restored by the PPCNTR instruction, assuming that the intervening instructions do not alter the subroutine stack.

6.2.5.6 SETS

Set Sign Bit

$SR_1 \leftarrow 1$

$Y \leftarrow PC + 1$

Description:
Sets the sign bit (bit 1 in the status register). The sign bit normally contains a copy of the sign bit of the most recently modified counter.

Opcode:
```
011 0100
```

Example:
```
1020    101 01 01       JTWO
1021    011 01 00       SETS
            |
        (instructions)
            |
102A    111 11 00       JSA         H#1000
```

The SR_1 bit is set by the SETS instruction, which is executed only if the Flag input is HI at the JTWO instruction. The Flag status is thereby stored in SR_1. The subsequent JSA instruction is conditioned on this stored Flag status, because the Sign condition tests the SR_1 bit. Note that this scheme works only if SR_1 was not already HI before the JTWO instruction and no counter operations intervene between the SETS and JSA instructions.

6.2.5.7 WRCNTR

Transfer From Data Port to Counter

$C_i \leftarrow D$

$Y \leftarrow PC + 1$

Description:
Loads the specified counter from the data port.

Opcode:

```
011 10 ii
```

Example:

```
1000    011 10 11       WRCNTR      H#0015
```

The C_3 counter is initialized with H#0015.

6.2.6 Status Register Instructions

Status register instructions affect the entire contents of the status register. Instructions that affect the individual bits of the status register are listed in other sections according to the function the bit or bits perform. The status register can be pushed to or popped from the subroutine stack; it can also be loaded or read from the data port.

PPSR	Pop Subroutine Stack To Status Register
PSSR	Push Status Register To Subroutine Stack
RDSR	Read Status Register At Data Port
WRSR	Write Status Register From Data Port

6.2.6.1 PPSR

Pop Subroutine Stack to Status Register

SR ← SS

Y ← PC + 1

Description:
Loads the status register with the value of the stack location specified by the SSP and decrements the SSP.

Opcode:
```
010 0010
```

Example:
```
1000    010 0001        PSSR
1001    001 1100        WRSR        H#FFC0
            |
        (instructions)
            |
1010    010 0010        PPSR
```

This example overwrites the status register but restores the old value later. The PSSR instruction pushes the status register on the subroutine stack. The WRSR instruction loads the status register with a new value (H#FFC0). Later, the PPSR instruction restores the old status register value, assuming that the intervening instructions do not change the subroutine stack.

6.2.6.2 PSSR

Push Status Register to Subroutine Stack

SS ← SR

Y ← PC + 1

Description:
Increments the SSP and loads the stack location specified by the SSP with the status register value.

Opcode:

```
010 0001
```

Example:

```
1000    010 0001        PSSR
1001    001 1100        WRSR        H#F003
                |
        (instructions)
                |
1010    010 0010        PPSR
```

This example overwrites the status register but restores the old value later. The PSSR instruction pushes the status register on the subroutine stack. The WRSR instruction loads the status register with a new value (H#F003). Later, the PPSR instruction restores the old status register value, assuming that the intervening instructions do not change the subroutine stack.

6.2.6.3 RDSR

Transfer From Status Register to Data Port

D ← SR

Y ← PC + 1

Description:
Outputs the value of the status register (SR_{15-0}) on data port (D_{15-0}).

Opcode:

```
010 1110
```

Example:

```
1000    010 1110        RDSR
```

The status register value is transferred to the data port, allowing another device to read it.

6.2.6.4 WRSR

Transfer Data Port to Status Register

SR ← D

Y ← PC + 1

Description:
Loads the status register, SR_{15-0}, from the data port, D_{15-0}.

Opcode:

```
001 1100
```

Example:

```
1000    001 1100        WRSR        H#E05C
```

The WRSR instruction loads the value H#E05C into the status register, which sets options as follows:

H#E05C = 1110 0000 01 01 1 1 0 0

- 0 — Interrupt Latching Enabled
- 0 — Sign Bit Cleared
- 1 — Interrupts Enabled
- 1 — GSP Selected
- 01 — 8-Bit Relative Offset Width
- 1110 0000 01 — IR_{9-7} and IR_0 Masked

6.2.7 Relative Jump Offset Width Instructions

Relative jump offset width instructions set the values of two status register bits (SR_{5-4}) that control the width of the offset value read at the data port for relative jump instructions. The Instruction Hold Control (IHC) instruction, listed under Miscellaneous Instructions, also affects offset width.

REL16	Select 16-Bit Relative Jump Offset Width
REL12	Select 12-Bit Relative Jump Offset Width
REL8	Select 8-Bit Relative Jump Offset Width

6.2.7.1 REL16

Select 16-Bit Relative Jump Offset Width

$SR_5 \leftarrow 0$

$SR_4 \leftarrow 0$

$Y \leftarrow PC + 1$

Description:
Selects 16-bit relative jump offset widths by setting the values of $SR_{5\text{-}4}$ to 00. Relative jump instructions input offset values from all 16 data port bits ($D_{15\text{-}0}$).

Opcode:

```
010 0100
```

Example:

```
1000    010 0100        REL16
1001    111 1001        JDR         H#53C0
```

The REL16 instruction selects a 16-bit offset width, so that the JDR instruction adds the full 16-bit value (H#53C0) to the incremented program counter value (H#1002) to form the jump address (H#63C2) if Flag is HI.

6.2.7.2 REL12

Select 12-Bit Relative Jump Offset Width

$SR_5 \leftarrow 1$

$SR_4 \leftarrow 1$

$Y \leftarrow PC + 1$

Description:
Selects 12-bit relative jump offset widths by setting the values of SR_{5-4} to 11. Relative jump instructions input offset values from the 12 least significant data bits (D_{11-0}); D_{15-12} are ignored. The sign bit of the offset value is repeated to extend the offset internally to 16 bits before the offset is added to the program counter value.

Opcode:
```
010 0111
```

Example:
```
1000    010 0111        REL12
1001    111 1110        JSR         H#E3C0
```

The REL12 instruction selects a 12-bit offset width, so that in the JSR instruction, the four most significant bits of the data port (value H#E) are ignored. The offset value H#03C0 is added to the (incremented) program counter value (H#1002) to form the jump address (H#13C2) if the Sign condition is true.

6.2.7.3 REL8

Select 8-Bit Relative Jump Offset Width

$SR_5 \leftarrow 0$

$SR_4 \leftarrow 1$

$Y \leftarrow PC + 1$

Description:
Selects 8-bit relative jump offset widths by setting the values of SR_{5-4} to 01. Relative jump instructions input offset values from the eight least significant data port bits (D_{7-0}); D_{15-8} are ignored. The sign bit of the offset value is repeated to extend the offset internally to 16 bits before the offset is added to the program counter value.

Opcode:
```
010 0110
```

Examples:
```
1000    010 0111        REL8
1001    111 1110        JSR         H#E3C0
```

The REL8 instruction selects an 8-bit offset width, so that in the JSR instruction, the eight most significant bits of the data port (value H#E3) are ignored. The offset value H#FFC0 (H#C0 with sign-extension) is added to the (incremented) program counter value (H#1002) to form the jump address (H#0FC2) if the Sign condition is true.

```
1000    010 0101        IHC
                |
        (instructions)
                |
10FF    010 0110        REL8
```

SR_{5-4} are used to set the IHC (Instruction Hold Control) mode as well as relative offset width; therefore, one of the three offset width instructions must be used to disable the IHC mode. In this example, the IHC instruction enables the IHC mode. Later, the REL8 instruction disables the IHC mode without changing the offset width, because the default offset width in the IHC mode is eight bits.

6.2.8 Miscellaneous Instructions

Miscellaneous instructions perform various functions.

CONT	Continue
IDLE	Idle
IHC	Select Instruction Hold Control Mode, 8-Bit Relative Jump Offset Width
WCS	Initiate Writeable Control Store Operation

6.2.8.1 CONT

Continue

$Y \leftarrow PC + 1$

Description:
Performs no operation except for outputting the address from the program counter and incrementing it at the end of the cycle.

The first instruction of an interrupt service routine must always be a CONT to allow the ADSP-1401 to push the return address on the subroutine stack before starting to execute the service routine.

Opcode:

```
000 0000
```

Example:

```
1000    000 0000        CONT
1001    000 0000        CONT
1002    000 0000        CONT
```

Three microcode instructions are executed in sequence. Repeated execution of CONT drives sequential program execution. Activity may be occurring at other devices.

6.2.8.2 IDLE

Disable Address Output and Suspend Program Execution

Y ← High Impedance

PC ← PC

Description:
Places the address port in the high impedance state and holds the program counter at its current value for one cycle. This instruction allows another device to address the microcode memory. IDLE is normally used in the IHC mode to repeat the IDLE instruction for several cycles. The IDLE instruction can not be repeated in microcode to hold the ADSP-1401 off the address bus. A series of IDLE instructions in microcode causes the ADSP-1401 to alternately place its address port in the high-impedance state and output the next sequential address, creating a potential for bus contention or execution of erroneous instructions.

External interrupt requests must be inhibited while IDLE is being executed. If interrupts are not inhibited, the ADSP-1401 will attempt to process an interrupt that goes active. However, it will be unable to output an interrupt vector because the IDLE instruction places the address port in the high-impedance state; more important, it will set its IRIP flag, which will inhibit further interrupt processing even after the IDLE state is exited. Interrupt processing can be inhibited using the interrupt mask or the DISIR instruction. While processing is inhibited, interrupt requests will still be latched.

Opcode:
001 0000

Example:
See example under IHC (next page).

6.2.8.3 IHC

Select Instruction Hold Control Mode

$SR_5 \leftarrow 1$
$SR_4 \leftarrow 0$
$Y \leftarrow PC + 1$

Description:
Selects the IHC (Instruction Hold Control) mode by setting the values of SR_{5-4} to 10. In the IHC mode, the IR_1 interrupt is redefined to activate an instruction hold; the instruction that is latched when IR_1 is asserted is repeated until IR_1 is deasserted. Assertion and deassertion of IR_1 must meet the input setup requirement (see the ADSP-1401 Data Sheet). IR_1 must not be asserted with the IHC instruction, because this would cause the ADSP-1401 to repeat the IHC instruction, ignoring the instruction port and outputting sequential addresses indefinitely.

Because the SR_{5-4} bits also control relative jump offset width, the width setting defaults to eight bits in the IHC mode.

Opcode:
```
010 0101
```

Example:
```
1000    001 0110        DISIR
1001    010 0101        IHC
1002    001 0000        IDLE
1003    010 0100        REL16
1004    000 0001        CAIR
1005    011 0110        ENAIR
```

In this example, the IHC mode is used to repeat the IDLE instruction. The DISIR instruction is used here to disable interrupts explicitly for two reasons. First, IR_1 must continue to be disabled even after the IR_1 input is deasserted to prevent the IR_1 interrupt from being processed, because the IR_1 latch will be set. (If IR_1 is masked, this is not a problem.) Second, interrupts must be disabled to prevent an interrupt from activating the IRIP signal during the IDLE instruction. Because the interrupt vector cannot be output during the idle state, the service routine cannot be executed. IRIP will not be cleared and will inhibit interrupt processing even after the ENAIR instruction re-enables interrupts.

After the IHC instruction is executed, IR_1 is asserted during the IDLE instruction, which repeats as long as IR_1 is active. When IR_1 is deasserted, the REL16 instruction is executed to exit the IHC mode and also set 16-bit relative jump offset widths. The CAIR instruction clears the interrupt latch to clear IR_1; this step is not necessary if IR_1 is masked. Finally, the ENAIR instruction re-enables interrupt processing.

6.2.8.4 WCS

Download to Writeable Control Store

$Y \leftarrow D$
REPEAT
IF FLAG
THEN $Y \leftarrow PC + 1$, $C_0 \leftarrow C_0 - 1$
ELSE $Y \leftarrow PC$

Description:
Initiates a download operation to writeable control store. The operation consists of two parts: the first part is a jump to the address supplied at the data port; the second part increments and outputs the address and decrements C_0 (if the Flag input is HI) or inhibits the program counter increment (if the Flag input is LO). The second part is repeated indefinitely and is terminated by an external interrupt (IR_{8-1}), counter underflow interrupt (IR_0) or reset. C_0 should be preloaded before the WCS instruction is executed if C_0 underflow is used to terminate the download. If C_0 underflow is not used, IR_0 should be disabled (masked).

Opcode:
```
010 0000
```

Example:
```
0000    000 0000        CONT
0001    011 1000        WRCNTR      H#7FFE
0002    001 1101        SLRIVP      H#0020
0003    000 1101        WRIV        H#0006
0004    011 0110        ENAIR
0005    010 0000        WCS         H#0006
```

These six instructions, which occupy locations H#0000-0005 in microcode, program the ADSP-1401 to provide addresses to download 32K (H#8000) instructions and begin execution at the first downloaded instruction. This example uses the counter (C_0) underflow interrupt to terminate the WCS operation.

The CONT instruction is at location H#0000 because the first instruction after reset must be a CONT. The WRCNTR loads C_0 with the initial count value (H#7FFE), which is two less than the number of instructions to download (because the ADSP-1401 outputs one address after the counter has underflowed and one before it outputs the interrupt vector). The SLRIVP instruction initializes the IVP to location 0. The WRIV instruction loads the value H#0006 into interrupt vector location 0. H#0006 is the address of the first instruction to be executed after the download. When C_0 underflows, this address value will be output, and execution will continue from there. The ENAIR instruction is executed to enable interrupt processing, because the default state after reset disables interrupt processing. The WCS instruction initiates the download.

The first instructions executed after the download must clear the IR_0 interrupt, the IRIP signal, and remove the return address from the subroutine stack.

6 Instruction Set

6.3 MNEMONIC AND OPCODE SUMMARY

This section is a quick reference to ADSP-1401 opcodes and standard mnemonics. The instructions are grouped by function. Within each group, the instructions are listed alphabetically. The 2-bit codes *cc* and *ii* are defined as follows:

cc	*Condition*	*ii*	*Index*
00	Unconditional	00	0
01	Not Flag	01	1
10	Flag	10	2
11	Sign	11	3

Status register bit definitions are summarized below:

Bit	*HI(1)*	*LO(0)*
0	Transparent Interrupts	Latched Interrupts
1	Sign bit of the most recently decremented counter (*before* decrement)	
2	All Interrupts Enabled	All Interrupts Disabled
3	GSP Selected	LSP Selected

5 - 4 Relative Jump Offset Width and IHC Mode Selection

11	12 Bits
10	IHC Mode, 8 Bits
01	8 Bits
00	16 Bits

15 - 6 Interrupt Mask (Bit 15 masks IR_9, 14 masks IR_8, and so on)

HI(1)	*LO(0)*
Interrupt Disabled	Interrupt Enabled

Conditional Jump and Branch

BRANCH	100 ccii	If Sign Jump Register Else If Condition Jump Data, Absolute (Condition ≠ Sign)
JDA	111 cc11	If Condition Jump Data, Absolute
JDI	101 cc10	If Condition Jump Data, Indirect
JDR	111 cc01	If Condition Jump Data, Relative
JDRST	100 11ii	If Sign Jump Data, Absolute, and Reset Counter
JPCNF	011 0101	If Not Flag Jump PC
JPCOF	001 0101	If Flag Jump PC
JRC	110 ccii	If Condition Jump Register (Condition ≠ Sign)
JRS	110 11ii	If Sign Jump Register and Decrement Counter
JSA	111 cc00	If Condition Jump Subroutine, Absolute
JSR	111 cc10	If Condition Jump Subroutine, Relative
JTWO	101 cc01	If Condition Jump PC + 2
RTN	101 cc11	If Condition Return From Subroutine

Subroutine Stack

DSSP	000 0010	Decrement SSP By 1
PPSSD	011 1110	Pop Subroutine Stack To Data Port
PSDSS	001 1110	Push Data Port To Subroutine Stack
RDSSP	010 1100	Read SSP At Data Port
WRSSP	000 1110	Write SSP From Data Port

Instruction Set 6

Interrupt Control

CAIR	000 0001	Clear All Interrupts
CCIR	001 0001	Clear Current Interrupt
DISIR	001 0110	Disable All Interrupts
ENAIR	011 0110	Enable All Interrupts
IRMBC	001 0011	Interrupt Mask Bitwise Clear
IRMBS	001 0010	Interrupt Mask Bitwise Set
RDIV	010 1101	Read Interrupt Vector At Data Port
RTNIR	000 0011	Return From Interrupt Routine
SLIR	001 0111	Select Latched Interrupts
SLRIVP	001 1101	Write SLR and IVP From Data Port
STIR	011 0111	Select Transparent Interrupts
WRIV	000 1101	Write Interrupt Vector From Data Port

Register Stack

AIRSP	010 10ii	Add i to RSP (i = 00 = Add 4)
PPGSP	000 0100	Pop Subroutine Stack To GSP
PPRSD	011 1111	Pop Register Stack To Data Port
PSDRS	001 1111	Push Data Port To Register Stack
PSGSP	000 0101	Push GSP To Subroutine Stack
PSPC	010 0011	Push Program Counter To Register Stack
RDRSP	010 1111	Read RSP At Data Port
S1RSP	000 1111	Subtract 1 From RSP
S4RSP	011 1100	Subtract 4 From RSP
SGSP	000 0111	Select GSP
SLSP	000 0110	Select LSP
WRRSP	000 1100	Write RSP From Data Port

Counter

CLRS	001 0100	Clear Sign Bit (SR_1)
DCCNTR	011 00ii	Decrement Counter By 1
IFCDEC	101 cc00	If Condition Decrement C_0
PPCNTR	001 10ii	Pop Subroutine Stack To Counter
PSCNTR	000 10ii	Push Counter To Subroutine Stack
SETS	011 0100	Set Sign Bit (SR_1)
WRCNTR	011 10ii	Write Counter From Data Port

Status Register

PPSR	010 0010	Pop Subroutine Stack To Status Register
PSSR	010 0001	Push Status Register To Subroutine Stack
RDSR	010 1110	Read Status Register At Data Port
WRSR	001 1100	Write Status Register From Data Port

Relative Jump Offset Width

REL16	010 0100	Select 16-Bit Relative Jump Offset Width
REL12	010 0111	Select 12-Bit Relative Jump Offset Width
REL8	010 0110	Select 8-Bit Relative Jump Offset Width

Miscellaneous

CONT	000 0000	Continue
IDLE	001 0000	Idle
IHC	010 0101	Select IHC Mode, 8-Bit Relative Jump Offset Width
WCS	010 0000	Initiate Writeable Control Store Operation

Internal Architecture 7

7.1 INTRODUCTION

The ADSP-1410 is a 48-pin CMOS device. The names and definitions of its pins are listed in Table 7.1.

PIN NAME	*DEFINITION*
I_{9-0}	Instruction Input, 10 bits
Y_{15-0}	Address Output, 16 bits
D_{15-0}	Data I/O, 16 bits
DSEL	Data Selection Control Input
AIRE	Alternate Instruction Register Enable Input
CMP/Z	Two-Function (COMPARE and ZERO) Flag Output
CLK	Clock Input
V_{DD}	+5 Volt Power Supply
GND	Ground

Table 7.1 Pin Definitions

Figure 7.1 shows a block diagram of the ADSP-1410. The device consists of the following major areas, which are described in the sections of this chapter:

- Instruction Port and Instruction Decoder
- Alternate Instruction Register
- Bidirectional Data Port
- Address (R) Registers
- Offset or Base (B) Registers
- Arithmetic Logic Unit (ALU) and Shifter
- Compare (C) Registers and Initialization (I) Registers
- Address Port and Bit Reverser
- Control Register

7 Internal Architecture

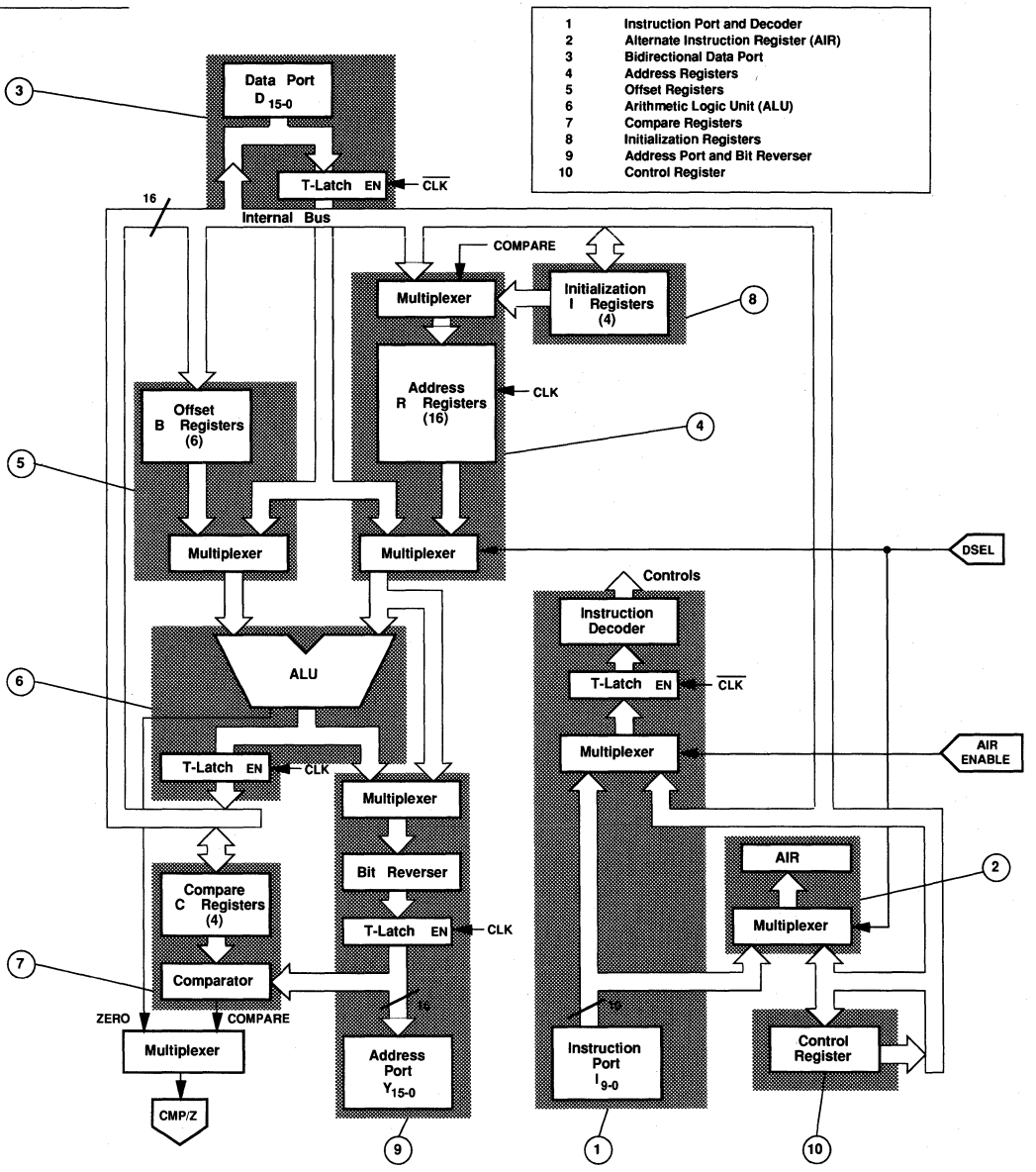


Figure 7.1 ADSP-1410 Block Diagram

Internal Architecture 7

7.2 INSTRUCTION PORT

The instruction port consists of 10 pins (I_{9-0}) through which the ADSP-1410 receives its 10-bit instruction. The instruction port is latched during clock HI. During clock LO, the instruction latch is transparent, so that the ADSP-1410 can begin decoding the next instruction as soon as it becomes available from microcode. Figure 7.2 shows the instruction latch timing.

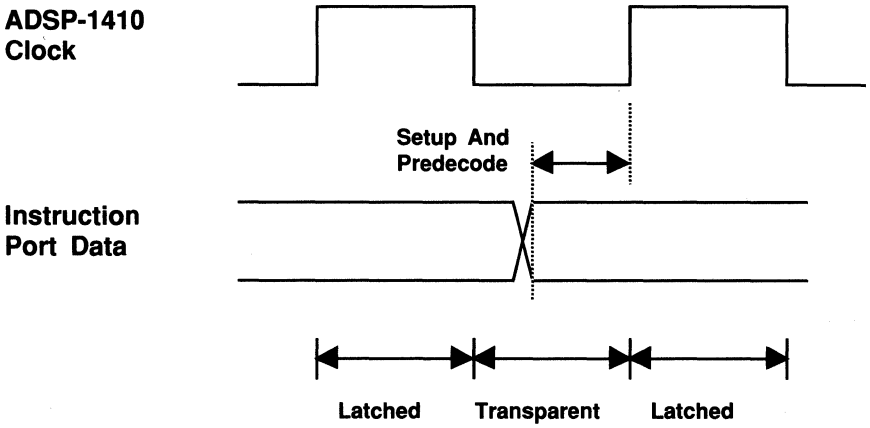


Figure 7.2 Instruction Latch Timing

7.3 ALTERNATE INSTRUCTION REGISTER (AIR)

The Alternate Instruction Register (AIR) is a 10-bit register that provides an alternate instruction source to the instruction latch. The Alternate Instruction Register Enable (AIRE) input pin controls an internal multiplexer that selects either the instruction port or the AIR. When the AIRE input is active (HI), the AIR contents are transferred to the instruction latch and the instruction port is ignored. The timing requirements for the AIRE input are the same as for the instruction port.

The AIR allows the execution of an internally-stored instruction in place of an instruction from microcode, under control of the AIRE input. The ADSP-1410 can also operate in a mode in which the AIR is enabled conditionally by the internal comparator. For more information, see *Alternate Instruction Register* in Chapter 8.

7 Internal Architecture

7.4 BIDIRECTIONAL DATA PORT

The data port (D) consists of 16 bidirectional pins (D_{15-0}). The ADSP-1410 loads data from the port into internal registers or directly to the internal ALU; it also outputs the contents of internal registers to other devices through the data port. Data input is latched during clock HI. Data output is driven while the clock is HI and disabled while the clock is LO, allowing time for external data setup if the next cycle requires data input. This timing is shown in Figure 7.3. The output drivers are placed in the high-impedance state when the ADSP-1410 is not writing data.

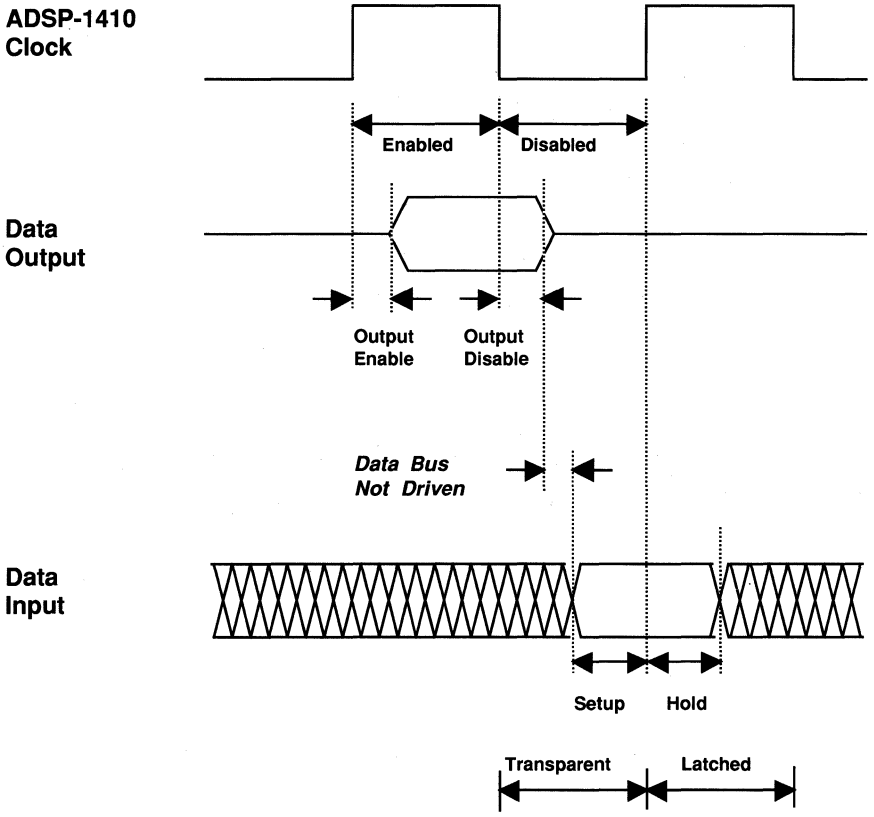


Figure 7.3 Data Port Timing

7.5 ADDRESS REGISTERS

The 16 address registers (R_{15-0}) store 16-bit data memory addresses. These registers enable you to keep track of 16 different pointers. Certain addressing instructions output and update the address in an R register. The update mode, which you select with an overhead instruction (see *Control Register*, below), determines whether the address is output before or after it is updated.

The data port can also provide the address to output, as shown in Figure 7.4. Address updates are always written to the R register specified in the instruction opcode, whether the address source is an R register or the data port.

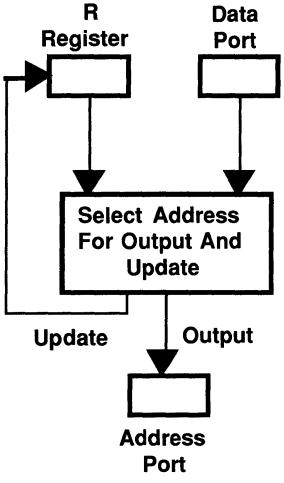


Figure 7.4 Address Output and Update Paths

In most instructions, four opcode bits specify one of the 16 R registers. Two instructions (YADD and YSUB) use three bits of opcode plus one control register bit to specify the R register. For these instructions, the R registers are grouped into two banks of eight registers each (R_{15-8}, R_{7-0}), and the control register bit (see *Control Register*, below) selects the bank. This bank selection bit can be set or cleared through a dedicated instruction.

7.6 OFFSET REGISTERS

The six offset (base) registers (B_{6-4}, B_{2-0}) store 16-bit numbers used to update addresses. Figure 7.5 shows the paths from the B registers to the ALU. The registers are grouped into two banks of three registers; "register" selections B_7 and B_3 cause the ADSP-1410 to read the offset value directly from the data port.

7 Internal Architecture

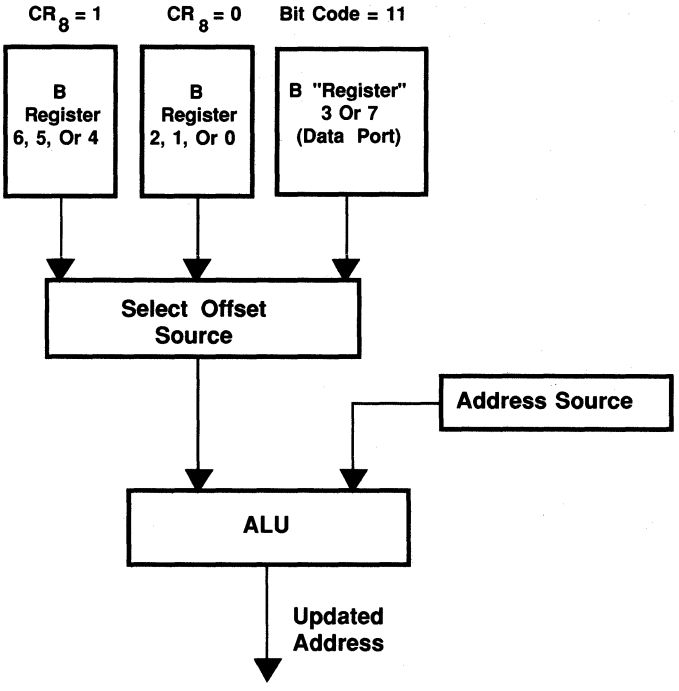


Figure 7.5 Offset Paths

A control register bit selects one of the two B banks. Instructions that specify a B register select one of the three registers or the data port through two bits of opcode, as shown in Table 7.2. Because either B_7 or B_3 selects the data port, the status of the B bank selection bit does not matter when the specified code is 11.

CR_8	*Two-Bit Code*	*Offset Source*
0	00	B_0
0	01	B_1
0	10	B_2
0	11	Data Port
1	00	B_4
1	01	B_5
1	10	B_6
1	11	Data Port

Table 7.2 B Register Selection

Internal Architecture 7

7.7 ARITHMETIC LOGIC UNIT (ALU) AND SHIFTER

The Arithmetic Logic Unit (ALU) and shifter can perform the following operations:

Two Operands	*One Operand*
Addition	Increment
Subtraction	Decrement
OR	Shift Left
AND	Shift Right
XOR	

In most two-operand instructions, the operands are provided by an R register and a B register. Alternatively, the data port can provide one of the operands. (In fact, the same data port value can be used for both operands.)

Left and right shifts are single-bit; left shifts are logical (vacated bit filled with zero) whereas right shifts are arithmetic (vacated bit filled with sign bit).

The ADSP-1410 detects whether the result of a logical operation (OR, AND, XOR) is zero and flags this condition on its Compare/Zero (CMP/Z) output.

7.8 COMPARE REGISTERS AND INITIALIZATION REGISTERS

Circular buffers require the ability to update and output an address repeatedly until the address value falls outside the buffer, then reset the address value to the beginning of the buffer. Four 16-bit compare registers (C_{3-0}) and the corresponding 16-bit initialization registers (I_{3-0}) are used by several ADSP-1410 instructions to perform this function. I registers store the starting addresses of buffers, and C registers store boundary values, as shown in Figure 7.6. The C register value may be an upper or lower boundary, depending on the direction in which address sequencing takes place. The ADSP-1410 comparator checks the address being output against a boundary value in a C register. If the address has reached or gone beyond the boundary, the value in the I register (that has the same index as the C register) is used to reinitialize the R register; otherwise, the R register is updated as dictated by the particular instruction.

The type of comparison performed (upper boundary or lower boundary) by the comparator depends on the instruction. The comparator asserts a compare flag HI to enable reinitialization. The compare flag is also sent to the Compare/Zero (CMP/Z) output pin.

7 Internal Architecture

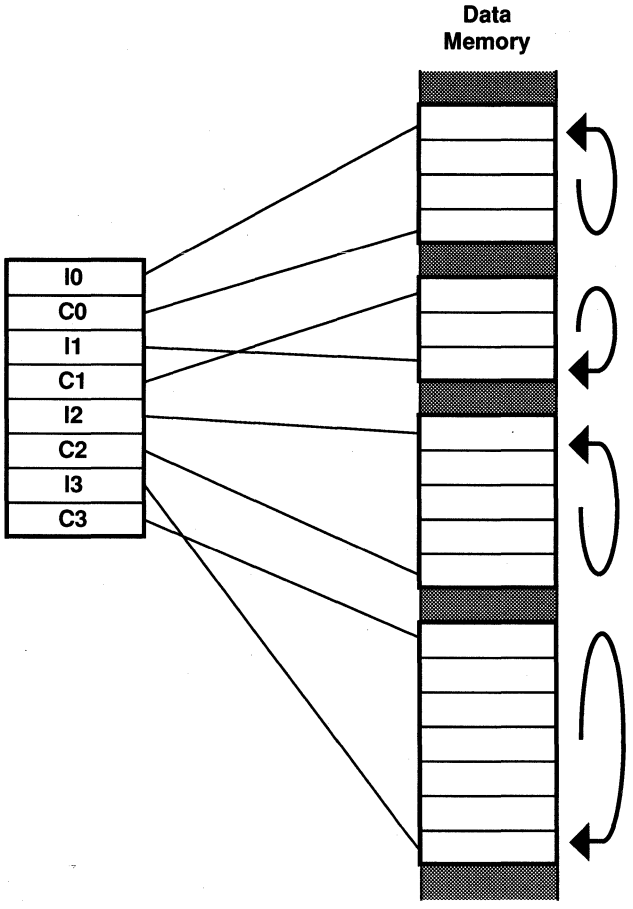


Figure 7.6 Comparison and Reinitialization

The input to the comparator is always the output address; the update mode determines whether this address is output before or after being updated. For more information on update modes, see *Update Modes* in Chapter 8.

I registers can be selectively enabled and disabled through a four-bit mask in the control register. The bit in the control register must be set to enable reinitialization with the I register of the same index. If the bit is not set, the R register is never reinitialized, regardless of the comparator status.

7.9 ADDRESS PORT AND BIT REVERSER

The address port (Y) consists of 16 outputs (Y_{15-0}). The address can be passed unchanged from one of the R registers or the data port to the address port; it can also be updated by the ALU before being output, depending on the update mode (see *Update Modes* in Chapter 8). In the latched output mode, the address outputs are latched during clock LO. In the transparent output mode, the address outputs are asynchronous to the clock, resulting in a shorter output delay. However, the address output is not held to the end of the cycle. For more information, see *Address Output Modes* in Chapter 10. When no address is being output, the address port is placed in the high impedance state.

The bit reverser transposes the bits of an address about the center, interchanging bits 15 and 0, bits 14 and 1, and so on, as shown in Figure 7.7. Bit-reversed addresses are used to place the results of a radix-2 fast Fourier transform (FFT) in sequential order.

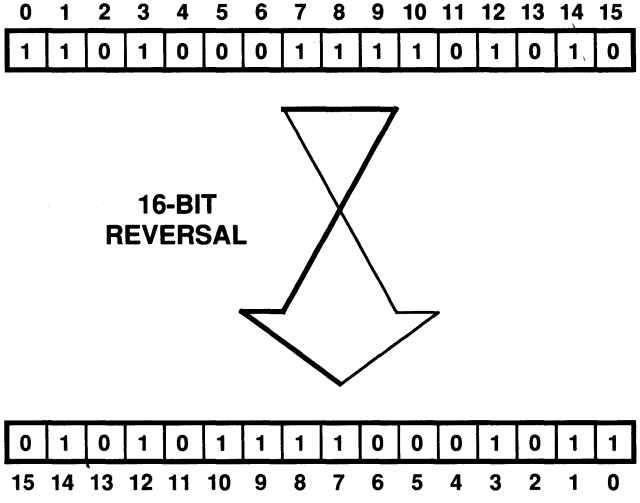


Figure 7.7 Bit Reversal

7.10 CONTROL REGISTER

The control register (CR), shown in Figure 7.8, consists of 11 bits that set the user-controlled options of the ADSP-1410. These options and their corresponding bits are described briefly in Table 7.3; details can be found in the indicated sections of this manual. The entire control register can be loaded from the 11 least significant bits of the data port. Individual bits can be set and cleared through dedicated instructions.

7 Internal Architecture

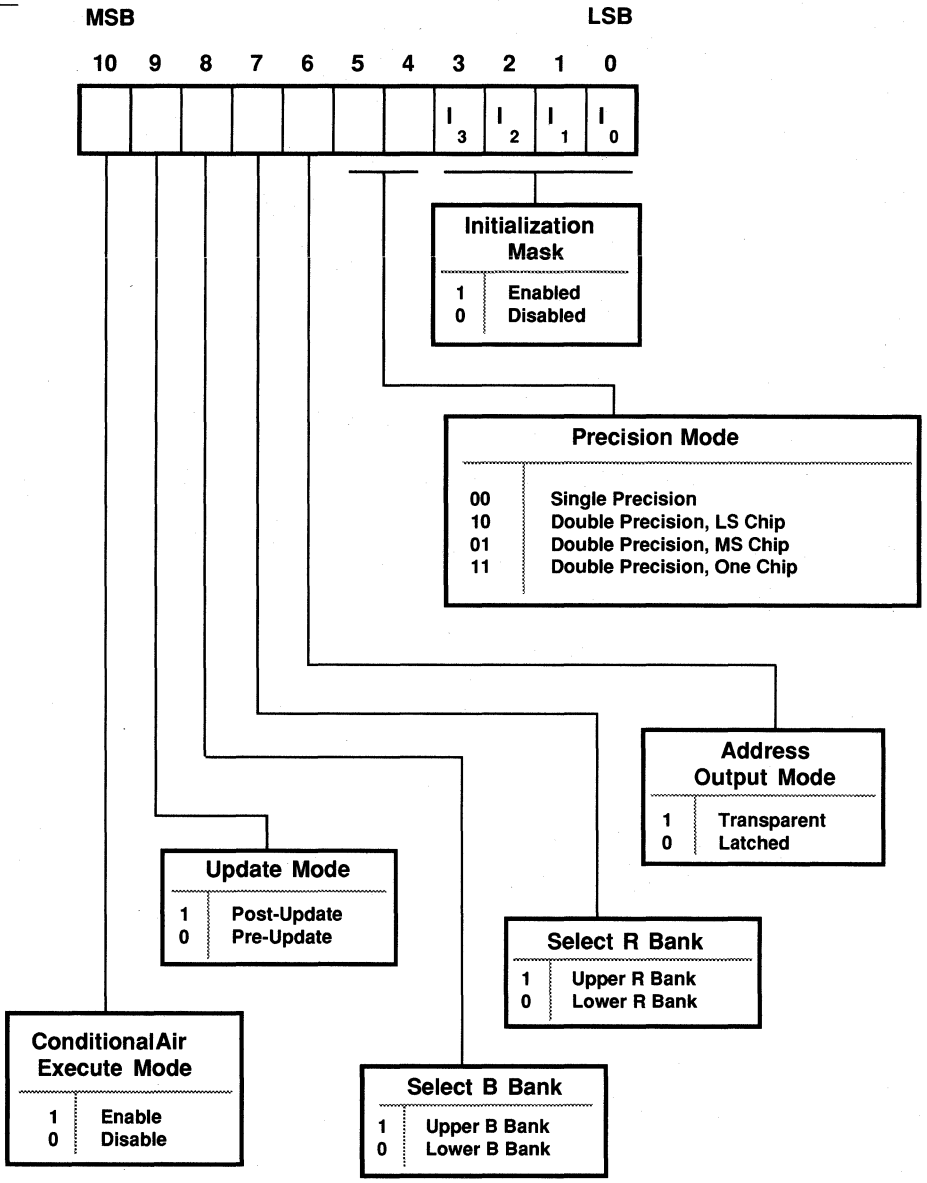


Figure 7.8 Control Register

Bit	Description
3 - 0	*Reinitialization Mask:* Each bit corresponds to the C register/I register pair of the same index number. If the bit is set, the corresponding I register is enabled, allowing conditional reinitialization of an R register with the contents of the I register. If the bit is cleared, reinitialization with the corresponding I register is disabled. See *Looping Instructions* in Chapter 8.
5 - 4	*Precision Mode Selection:* These bits contain a 2-bit code that selects one of four precision modes. The precision mode determines the width of the addresses the ADSP-1410 generates (30-bit or 16-bit) and the configuration in which 30-bit addresses are produced; see Chapter 9, *Precision Modes.*
6	*Transparent Mode Selection:* This bit, if set, disables the address latching that normally occurs in phase two (clock LO) of the cycle. In transparent mode, the address outputs change in response to changes at the instruction port; they are not synchronized to the clock. See *Address Output Modes* in Chapter 10.
7	*R Bank Selection:* The 16 address (R) registers are grouped into two banks of eight registers. For instructions that select one of eight registers (YADD and YSUB), this bit selects the upper bank (if the bit is set) or the lower bank (if the bit is cleared).
8	*B Bank Selection:* The six offset (B) registers are grouped into two banks of three registers plus the data port, which acts as a fourth "register" in each bank. This bit selects the upper bank (if the bit is set) or the lower bank (if the bit is cleared).
9	*Post-Update Mode Selection:* This bit selects the update mode for instructions that both update and output an address. If this bit is set, addresses are output after being updated (post-update mode). If this bit is cleared, addresses are output before being updated (pre-update mode). See *Update Modes* in Chapter 8.
10	*Conditional AIR Execute Mode Selection:* This bit selects conditional AIR execution. If this bit is set, an instruction that compares the output address with a C register will conditionally enable the AIR (on the following cycle) rather than conditionally reinitialize with an I register. This mode disables reinitialization with the I registers, regardless of the values in the initialization mask (CR_{3-0}). If this bit is cleared, conditional AIR execution is disabled, and reinitialization is gated by CR_{3-0}. See *Alternate Instruction Register* in Chapter 8.

Table 7.3 Control Register Bit Description

Addressing Operations 8

8.1 INTRODUCTION

An ADSP-1410 addressing instruction performs two basic operations: the output of a data memory address and the update of the existing address. This chapter describes variations of these basic operations through which you can program the ADSP-1410 to generate a particular sequence of addresses. Several user-controlled features are described:

- *Update Mode* determines the order in which address output and address update is performed.
- *Looping Instructions* have the added capability of reinitializing the address value if the update operation causes the address to reach or exceed a user-defined limit.
- *Data Selection* through the DSEL input enables the ADSP-1410 to output and update an address value directly from the data port.
- *Alternate Instruction Register (AIR)* provides a single instruction that the ADSP-1410 can execute instead of the instruction from microcode. This register is enabled unconditionally by a hardware input or conditionally by a looping instruction.

8.2 UPDATE MODES

For instructions that both output and update an address, the update mode determines whether the address is output before or after being updated.

- In the pre-update mode, the address is output *before* being updated.
- In the post-update mode, the address is output *after* being updated.

You select the update mode by setting or clearing a bit in the control register. In either mode, the updated value is written to the R register unless the R register is reinitialized by the instruction.

The pre-update mode provides a shorter clock-to-address delay than the post-update mode because the updated address has already been calculated and stored in the R register by the previous access to the same R register. However, the post-update mode provides the shortest latency from the update of an address to its output; in the post-update mode, the address is output in the same cycle as it is updated, whereas in the pre-update mode, the updated address is not output until

the next cycle at the earliest. In the ADSP-1410, you can select the mode that best suits your system's requirements. For example, if one of the operands in the update operation is not available until the cycle in which the updated address is needed, you should use the post-update mode.

8.3 LOOPING INSTRUCTIONS

Many digital signal processing implementations require multiple accesses to a range of data memory locations. The addressing sequence that accomplishes these accesses "wraps around" from the highest address to the lowest address of the range, or vice versa, as illustrated in Figure 8.1. Looping instructions allow the ADSP-1410 to generate these looping address sequences without overhead penalty.

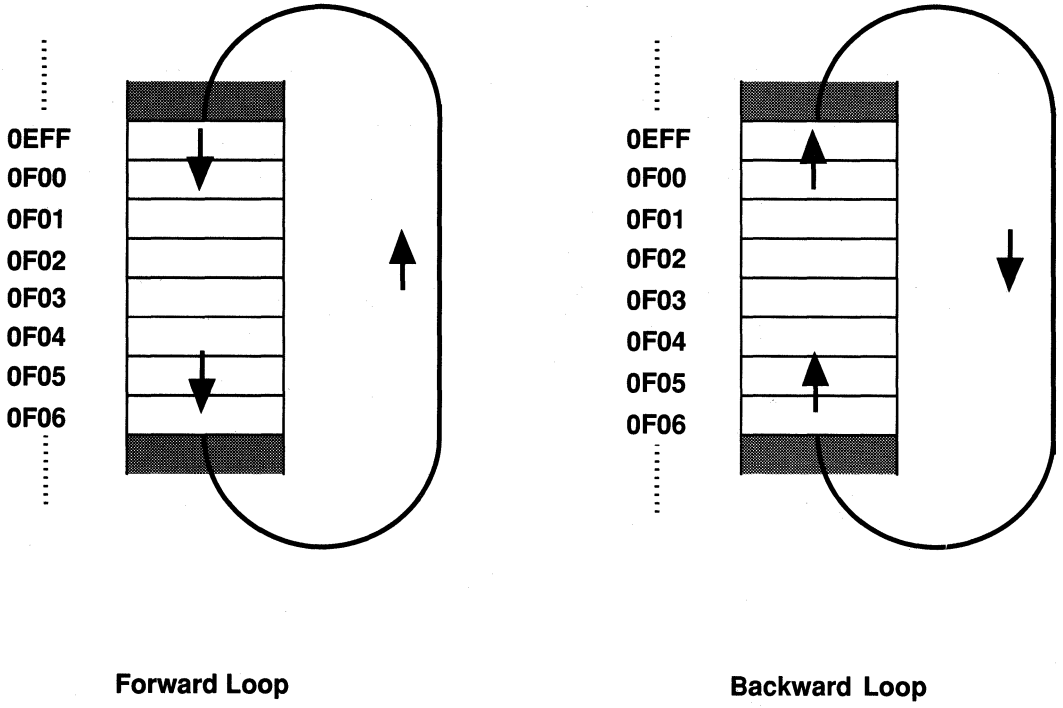


Figure 8.1 Looping Address Sequences

Addressing Operations 8

A looping instruction outputs address from an R register or from the data port and updates this address, in the order determined by the selected update mode. The address that is output is also compared to a boundary value, which is stored in a C register specified in the instruction. If the address has reached or passed the boundary value, the source R register is reinitialized with the value of the I register that has the same index as the C register. If the address has not reached the boundary value, the R register is written with the updated address.

Each C register/I register pair can store address boundaries, such as those used to implement a circular buffer. The circular buffer shown in Figure 8.2 loops forward from address H#0EFF to address H#0F06. I_2 stores the value H#0EFF and C_2 stores the value H#0F06. The loop is accomplished by iterating the YINC instruction, which updates the address in R_1 by incrementing it. When the value in R_1 matches the value in C_2 (H#0F06), R_1 is reinitialized to the starting address stored in I_2 (H#0EFF).

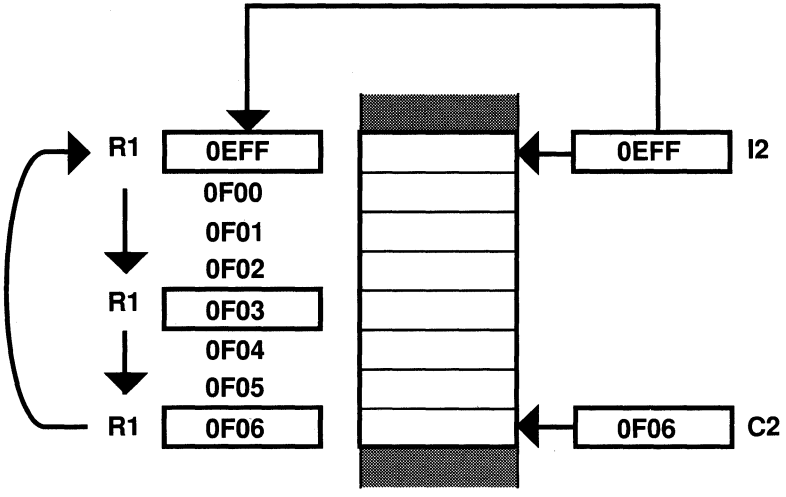


Figure 8.2 Reinitialization to Form Circular Buffer

8 Addressing Operations

The action of a looping instruction is summarized below. Note that four types of updates are possible (see Chapter 11 for instruction descriptions).

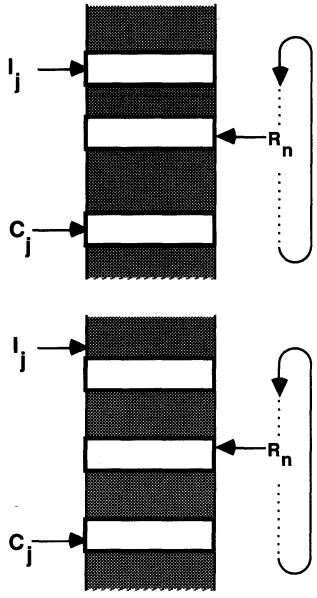
$Y \leftarrow R_n$ *or*	Transfer address in R_n to address port (pre-update mode), or
$Y \leftarrow R_n + (B_m, -B_m, 1, \text{ or } -1)$	Transfer updated address to address port (post-update mode).
IF $(Y \geq C_j)$ THEN $R_n \leftarrow I_j$	If output address equals or exceeds address in C_j, write address in I_j into R_n;
ELSE $R_n \leftarrow R_n + (B_m, -B_m, 1, \text{ or } -1)$	otherwise, write the updated address to R_n.

Y = address port
R_n = *nth* address register
B_m = *mth* offset register
C_j = *jth* compare register
I_j = *jth* initialization register

The internal comparator of the ADSP-1410 compares the output address to the value in the C register specified in the looping instruction. The comparison is unsigned; H#FFFF is greater than H#0000. The C register value is interpreted as either an upper or lower boundary, depending on whether the update performed in the instruction increases or decreases the address value. If the update will increase the value of the address, the C register value is interpreted as an upper limit; if the update will decrease the value of the address, the C register value is interpreted as a lower limit. For looping instructions that increase the address value (YINC and YADD), the compare flag is HI if the address value is greater than or equal to the C register value. For looping instructions that decrease the address value (YDEC and YSUB), the compare flag is HI if the address value is less than or equal to the C register value. A HI compare flag enables the appropriate I register for reinitializing the R register and is made available to other devices as well, through the Compare/Zero (CMP/Z) output (see *Compare/Zero* in Chapter 10).

Reinitialization can be masked (disabled) under user control, enabling you to switch between looping and linear address sequencing. One of four bits in the control register (CR_{3-0}) must be set to enable the I register with the same index for reinitialization. If the bit is cleared, no reinitialization is performed; the update performed by the particular instruction is written to the R register, regardless of the compare flag.

Because the update mode determines the address value that is output, it also affects the address that is compared with a C register to determine whether to reinitialize. Figure 8.3 shows two implementations of the same circular buffer, one using the pre-update mode and the other using the post-update mode. Both implementations use an update that increments the address by one.



Pre-Update Mode

$Y \leftarrow R_n$
IF $(Y \geq C_j)$ THEN $R_n \leftarrow I_j$
ELSE $R_n \leftarrow R_n + 1$

Post-Update Mode

$Y \leftarrow R_n + 1$
IF $(Y \geq C_j)$ THEN $R_n \leftarrow I_j$
ELSE $R_n \leftarrow R_n + 1$

Figure 8.3 Reinitialization in Pre-Update and Post-Update Modes

In the pre-update mode, R_n is output and then updated on each access. The update eventually increments R_n to the value of C_j, the last location of the buffer. The ADSP-1410 detects this condition when it outputs the value of R_n on the following access, and consequently reinitializes R_n to the value in I_j, which is the first location of the buffer.

In the post-update mode, R_n is updated first and then output on each access. When the update increments R_n to the value in C_j, the ADSP-1410 detects this condition in the *same* cycle and reinitializes R_n to the value in I_j, which in this case is the location before the start of the buffer. On the following access, this R_n value is incremented to the start of the buffer and then output.

Warning: In the pre-update mode, the updated address of a looping instruction is not compared with a boundary value until it is output by the next looping instruction that employs the same R register. However, the type of comparison ($\leq$ or $\geq$) performed by the next instruction depends on whether that instruction increases or decreases the address. If one instruction increases the address and a subsequent instruction decreases the address, the second instruction will not detect whether the address was increased beyond a boundary value because it tests for the address less than or equal to a boundary value. This situation is illustrated in Figure 8.4.

8 Addressing Operations

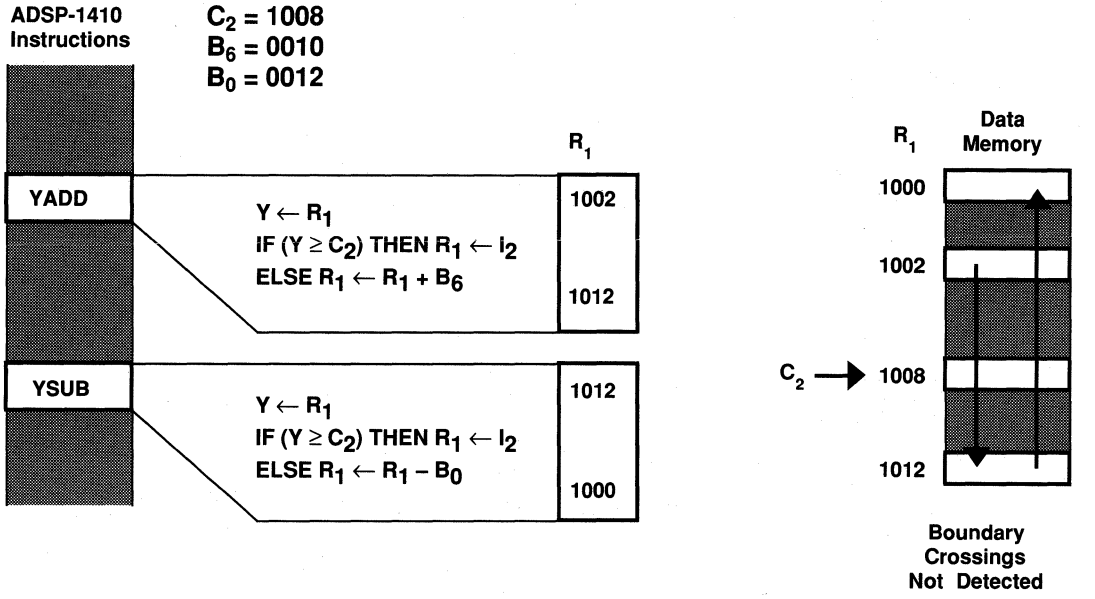


Figure 8.4 Address Increase Followed By Decrease In Pre-Update Mode

8.4 DATA SELECTION USING DSEL

For all instructions that output an address from an R register, you can assert the DSEL pin to replace the R register value with external data from the data port. The following instructions are affected by DSEL:

- Looping instructions (YADD, YSUB, YINC, YDEC)
- Logical instructions (YOR, YAND, YXOR)
- Shift instructions (YASR, YLSL)
- Register transfer instructions that drive the address port (YRTR, YRTB, YRTC)
- Bit reverse instructions (YREV)

When an active DSEL is asserted along with one of these instructions, the ADSP-1410 reads the address from the data port instead of the specified R register. If the address is updated, it is always written back to the R register. For example, in pre-update mode, the data port value is output to the address port, updated, and written to the specified R register. In post-update mode, DSEL allows the ADSP-1410 to read in a value from the data port, modify the value, and output the value at the address port in the same cycle, as shown in Figure 8.5. For register transfer instructions that use DSEL, the data port becomes the source register. You can load R, B, and C registers using this feature.

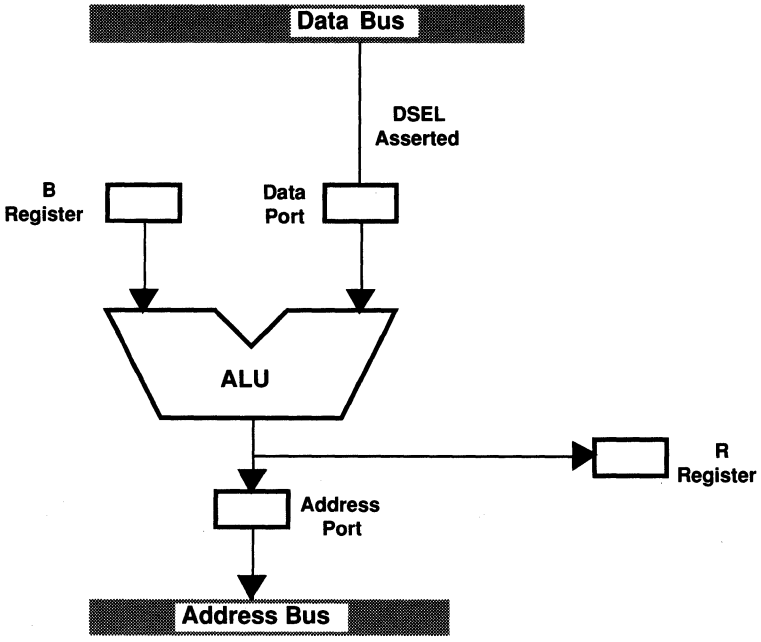


Figure 8.5 DSEL for Single-Cycle Input, Update, and Output

To read an offset value from the data port rather than a B register, you specify either B_7 or B_3 as the offset register; you do not assert DSEL because DSEL activates the data port as a replacement for only R registers. If you specify B_7 or B_3 and also assert DSEL, both the address and offset are input from the data port.

8.5 ALTERNATE INSTRUCTION REGISTER

The Alternate Instruction Register (AIR) is a 10-bit register which stores an instruction that is executed when the AIRE input is asserted. The AIRE input is latched and must meet the same setup and hold times as the instruction. The AIR is useful when circumstances require the repetitious or discriminate execution of an instruction. The AIR does not store the DSEL input. To activate the data port for an instruction executed from the AIR, you must assert DSEL along with AIRE.

If conditional AIR execution is enabled (CR_{10} is HI), the AIR is also activated by a true (HI) compare flag from a looping instruction. In conditional AIR execution mode, conditional reinitialization for looping instructions is disabled; the normal update operation is performed in all cases. In addition, if the compare flag is true, the AIR is enabled for the following cycle, superseding the next instruction at the instruction port.

8 Addressing Operations

One use of the AIR is to provide hardware tristate control to the address port, which is necessary to prevent address bus contention when several ADSP-1410s share the same address bus. Any instruction which does not output an address automatically places the address port in the high-impedance state. If you load such an instruction into the AIR, you can disable the address port by asserting the AIRE input. The NOP instruction is well suited for this application because it disables output without performing any operation.

Modulo addressing is one use of the conditional AIR execution mode. In this example, the ADSP-1410 is first loaded with the following values; then the YADD instruction (R_0, B_0 and C_2 specified) is executed repeatedly.

CR_9	0	Pre-Update Mode
CR_{10}	1	Conditional AIR Execution Mode
AIR	YSUB (R_0, B_1)	$Y \leftarrow R_0$; $R_0 \leftarrow R_0 - B_1$
C_2	3	Boundary
B_0	4	Step Size
B_1	7	AIR Update

The YADD instruction in the pre-update mode outputs the R_0 address and then compares its value with the value of C_2 (3). If the R_0 address is greater than or equal to 3, the AIR instruction (YSUB) is executed on the next cycle. Because conditional AIR execution is enabled, reinitialization is disabled, and thus R_0 is always updated by adding the offset stored in B_0 (4), whether or not it is greater than or equal to 3. If the AIR is enabled, YSUB outputs the R_0 address and subtracts the offset stored in B_1 (7) from R_0, yielding the modulo-11 value of the next address. This value is output on the next YADD.

When R_0 is initially 0, the addressing sequence illustrated in Figure 8.6 results. The first YADD instruction outputs 0, then adds 4 to R_0. The next YADD outputs 4, then adds 4 to R_0 and enables the AIR. YSUB outputs 8, then subtracts 7 from R_0, yielding 1. The rest of the addresses are generated in the same manner, wrapping around at the appropriate values. Single-cycle modulo addressing is thus accomplished without any overhead required to keep the address within the desired range.

In general, to perform modulo addressing in a circular buffer of length L, you initialize registers as listed below; R_n is any R register, B_x and B_y are any two B registers, and C_j is any C register:

R_n	Starting address *n*
B_x	Step size *m* (number of memory locations between consecutive data)
B_y	L–*m*
C_j	*n* + L–2*m* (pre-update mode) or *n* + L–*m* (post-update mode)
AIR	YSUB instruction (R_n and B_y specified as address and offset)

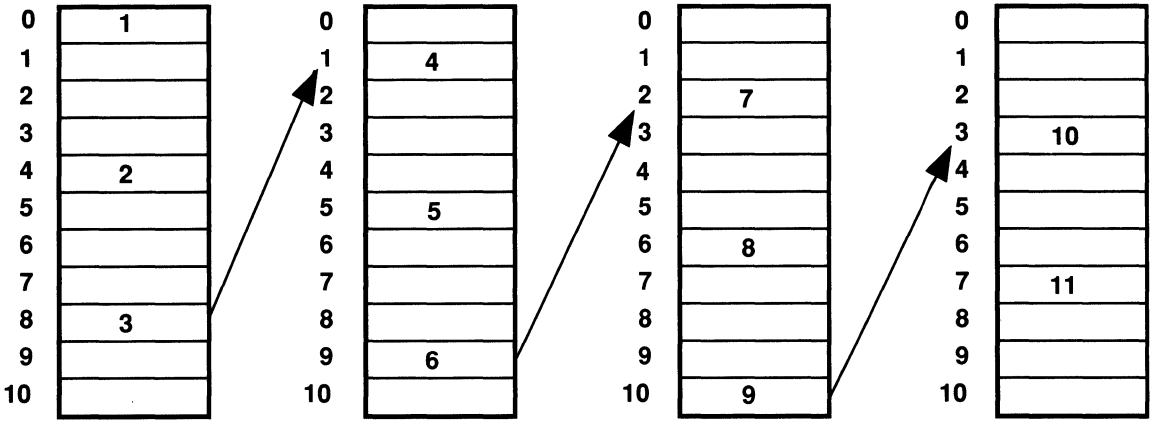


Figure 8.6 Modulo Addressing

You can load the AIR in one of two ways:

- *From The Data Bus:* To load the AIR from the data port, place the instruction on the 10 least significant data bits (D_{9-0}) and execute the WRA instruction.
- *Selectively, From The Instruction Bus:* To load the AIR from the instruction port, assert the DSEL input and execute the LDA instruction; on the next cycle, the ADSP-1410 loads the instruction at the instruction port into the AIR. DSEL must meet the same setup and hold times as the LDA instruction; otherwise, the instruction is not loaded. Therefore, if several ADSP-1410 devices share the same instruction space, you can load their AIRs selectively by asserting DSEL only for particular devices.

Because the RST instruction must take precedence over all other instructions, it is decoded separately from other instructions and cannot be executed from the AIR. If you try to executed a RST from the AIR, a NOP is performed instead. Furthermore, it is impossible to load the RST instruction into the AIR through the instruction port because an RST instruction at the instruction port is always executed, even on the cycle following a LDA instruction.

Precision Modes 9

9.1 INTRODUCTION

The precision mode controls the width of addresses generated by the ADSP-1410. Single precision provides 16 bits; double precision provides 30 bits (in two 15-bit words). In double-precision mode, one ADSP-1410 generates an address in two cycles or two cascaded ADSP-1410s generate an address in a single cycle.

Single precision provides the most flexibility in systems requiring no more than 16 address bits. The double-precision modes (one-chip and two-chip) provide 30 address bits but impose the following constraints:

- Neither double-precision mode supports bit reversal (the YREV instruction).
- The one-chip/double-precision mode requires two cycles to generate an address instead of one and does not support conditional reinitialization or conditional AIR execution.
- The two-chip/double-precision mode supports conditional reinitialization in the YINC and YDEC instructions, but not in the YADD and YSUB instructions. Conditional AIR execution is supported as usual.

The precision mode is determined by the values of bits 5 and 4 in the control register as follows:

CR_5	CR_4	*Mode*
0	0	Single precision
0	1	Double precision, least significant word (LSW)
1	0	Double precision, most significant word (MSW)
1	1	Double precision, one chip

The SETP instruction sets the values of these bits. This instruction is described in Chapter 11.

9.2 ONE-CHIP/SINGLE-PRECISION MODE

In single-precision mode, the address port includes all 16 address bits. Instructions execute in a single cycle, and all instructions are available.

9 Precision Modes

9.3 ONE-CHIP/DOUBLE-PRECISION MODE

In one-chip/double-precision mode, the lower 15 bits of the address port (Y_{14-0}) are address bits. With a single ADSP-1410, a 30-bit address can be generated in two cycles that output 15 bits each. Y_{15} is output HI if Y_{14-0} contain the MSW and LO if they contain the LSW. The Y_{15} output can be used to control an external multiplexer or latches to a 30-bit address bus, as shown in Figure 9.1.

With a single chip, all instructions that output an address must be executed twice, once for each 15-bit word. The ADSP-1410 automatically performs the operation on the LSW first, except in the right shift (YASR) instruction, in which the MSW is modified first. Only the 15 least significant bits of registers are used, and the most significant bit of the data port (D_{15}) is ignored. Address (R) registers must be paired (R_0 and R_1, R_2 and R_3, and so on) so that even-numbered registers hold LSWs and odd-numbered registers hold MSWs. Offset (B) and comparison (C) registers do not need to be paired. You can specify any B register or C register for either cycle of the double-precision operation.

Conditional reinitialization and conditional AIR execution in looping instructions are not supported in the one-chip/double-precision mode. Looping instructions become simple output-and-update operations. Unconditional execution of an instruction in the AIR is still supported. If the instruction needs to be repeated in order to generate a double-precision address, the AIRE input must be asserted for both cycles.

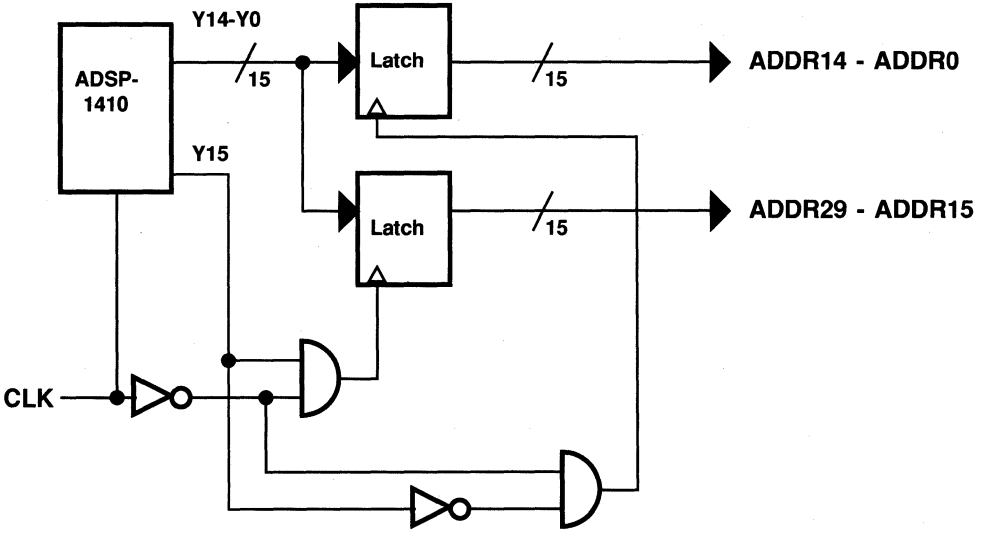


Figure 9.1 Single-Chip Connections to 30-Bit Address Bus

Although conditional reinitialization and conditional AIR execution is not supported in the one-chip/double-precision mode, the ADSP-1410 still performs the 30-bit comparison. It outputs the valid compare flag on the CMP/Z pin after the second cycle of the double-precision operation. The status of the CMP/Z pin after the first cycle is invalid and should not be recognized as a flag.

The ADSP-1410 performs 30-bit ALU operations by storing the carry or shifted bit resulting from the LSW operation and retrieving the bit for the next MSW operation. (In the case of a right shift, the shift bit resulting from the MSW operation is stored and retrieved for the next LSW operation.) Even if another instruction intervenes between the two operations, the double-precision operation can be completed, as long as the intervening instruction does not use the ALU.

9.4 TWO-CHIP/DOUBLE-PRECISION MODE

In the two-chip/double-precision mode, one ADSP-1410 generates only LSWs, the other only the corresponding MSWs. The connections between the two devices are shown in Figure 9.2. As in the one-chip/double-precision mode, only the lower 15 bits of the address port on each device are address bits. The Y_{15} pins of both devices are connected to one another in order to transfer carry, borrow, and shift bits from one device to the other.

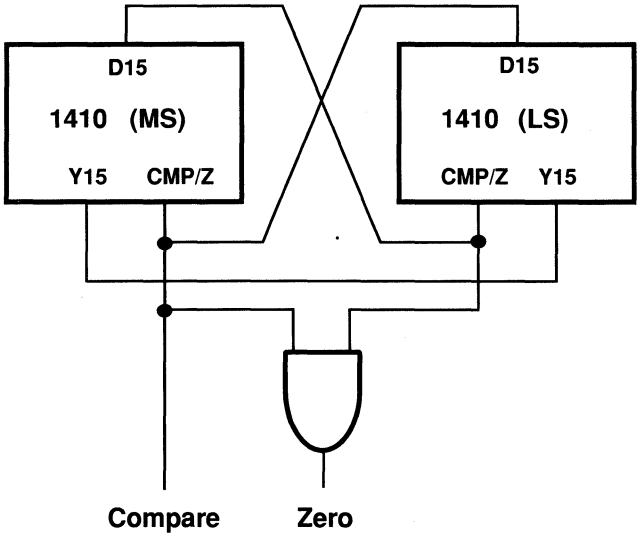


Figure 9.2 Two-Chip Cascade Connections

9 Precision Modes

Only the 15 least significant bits of any register and of the data port are used. The most significant bit of the data port (D_{15}) on each device receives the CMP/Z output from the other device. Therefore, D_{15} must not be connected to the data bus. The CMP/Z- to-D_{15} and Y_{15}-to-Y_{15} connections allow the ADSP-1410 to perform 30-bit comparisons, updating the address based on the combined compare status of both devices. The Compare flag for the double-precision operation is valid on the CMP/Z output of only the MSW chip, not the LSW chip. The Zero flag for the double-precision operation is the logical AND of the CMP/Z outputs of both chips.

In two-chip/double-precision mode, the two chips usually share the same instructions. The bits of the control register that determine the precision mode also determine whether the device generates the MSW or LSW. The SETP instruction, which sets the values of these bits, must set the bits on the MSW chip to the values that are the opposite of those on the LSW chip. Asserting DSEL on the MSW chip during the SETP instruction inverts the values of the control bits, thus allowing the same SETP instruction to be used to set the precision mode for both chips simultaneously.

Because of the time needed to transmit a carry/shift/borrow bit from one chip to the other, the two-chip/double-precision mode has an additional constraint on the minimum clock period. The maximum carry/shift/borrow output delay is referenced from the valid instruction input. The minimum carry/shift/borrow setup time is referenced to the falling edge of the clock. The sum of the delay and setup is the minimum time required from the valid instruction to the falling clock edge. Subtracting the minimum instruction setup time to the rising edge of the clock from this number yields the minimum clock HI time, as shown in Figure 9.3.

In the post-update mode, the address output delay is longer because the MSW must be calculated after the input of the carry/shift/borrow bit. The maximum MSW address output delay is referenced to the valid carry/shift/borrow bit. Because the address must be valid before the falling edge of the clock in order to be latched, this output delay is the determining factor in the minimum clock HI period. The sum of this delay and the maximum carry/shift/borrow delay from the LSW chip, minus the instruction setup time, determines the minimum clock HI time for post-update operations, as shown in Figure 9.3.

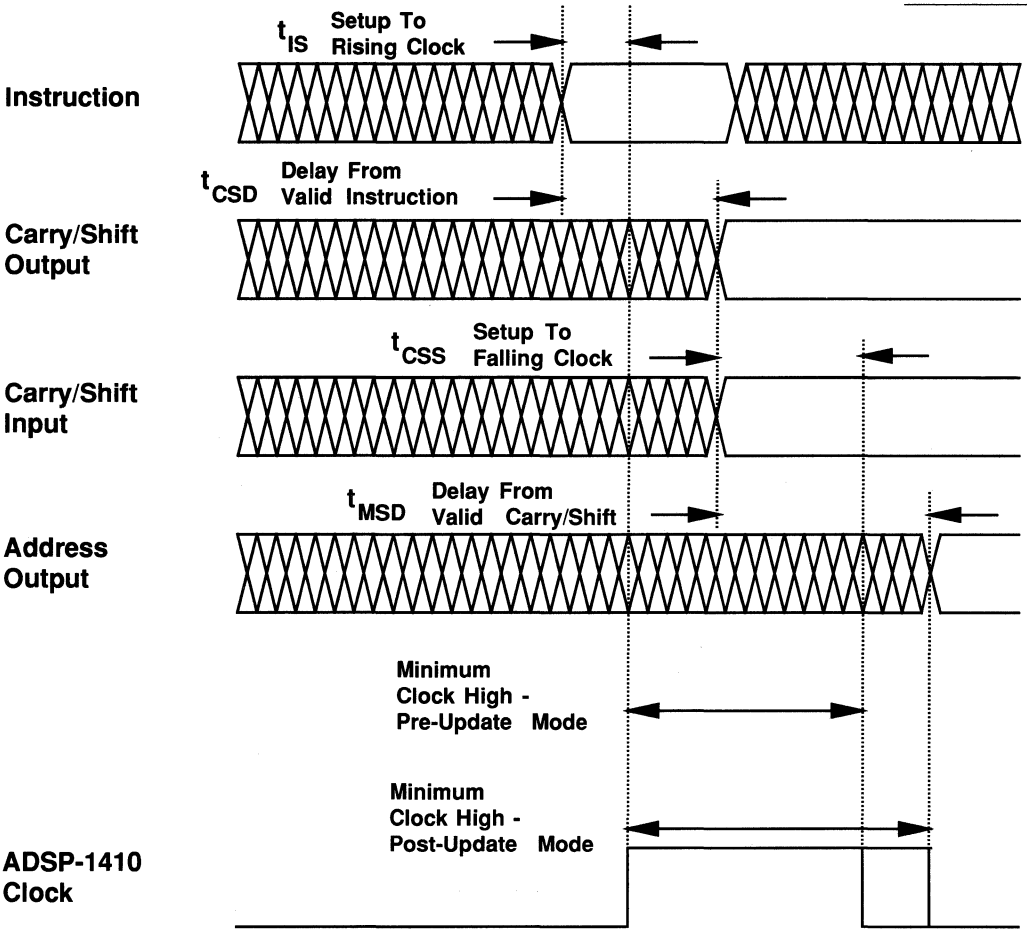


Figure 9.3 Minimum Clock Period For Two-Chip/Double-Precision Mode

System Interface 10

10.1 INTRODUCTION

The ADSP-1410 receives its instructions from microcode; it also receives data and control from microcode or from other devices in the system. It outputs addresses to the data memory address bus. Its dual-function CMP/Z flag may also be output to a monitoring device or circuit. These inputs and outputs determine the system interface requirements of the ADSP-1410.

10.2 LOOK-AHEAD PIPELINE

The Look-Ahead pipeline consists of two latching stages. The instruction port is latched during clock HI and is transparent during clock LO. In the normal (latched) output mode, the address port is transparent during clock HI and latched during clock LO. The address becomes valid in clock HI and is stable because the instruction is latched. When the instruction latch becomes transparent during clock LO, the address latch holds the address for the rest of the cycle. The instruction for the next cycle can be decoded at the same time the address for the current cycle is being held steady, as shown in Figure 10.1.

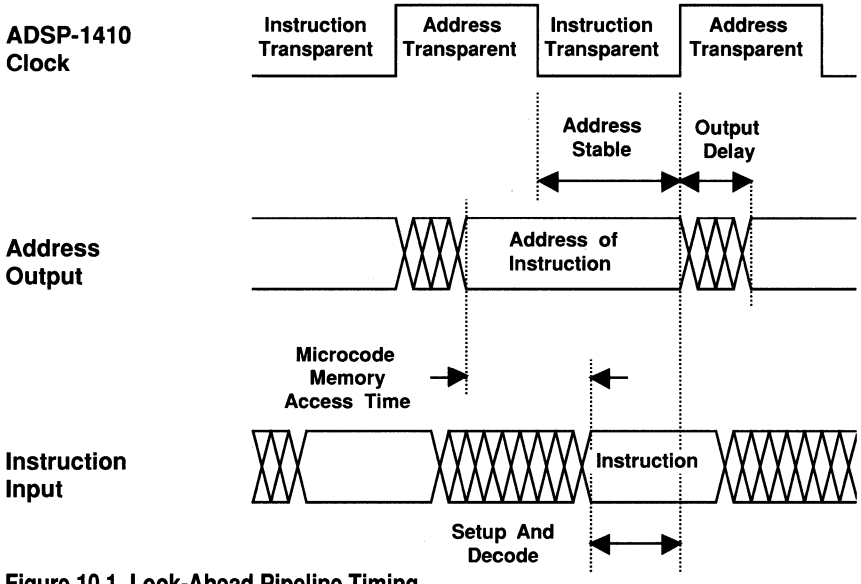


Figure 10.1 Look-Ahead Pipeline Timing

10 System Interface

The Look-Ahead pipeline provides two major advantages:

- No external instruction register or address register is needed. The ADSP-1410 can connect directly to microcode memory and data memory.
- Because the next instruction is partially decoded before the start of the next cycle, the address can be output sooner, allowing more time for the memory access.

10.3 DATA TRANSFERS

You can load and read all of the following registers through the data port using the following instructions:

REGISTER	*TO LOAD*	*TO READ*
Address (R) registers	YRTR* or BTR†	RTD
Offset (B) registers	YRTB*	BTD
Comparison (C) registers	YRTC*	CTD
Initialization (I) registers	DTI	ITD
Alternate Instruction Register (AIR)	WRA	RDA
Control Register (CR)	DTCR	CRTD

* DSEL asserted
† B_3 or B_7 register selected

Data can be input to the ALU from the data port either as an address or an offset. If DSEL is asserted during an instruction that normally loads an address from an R register, the data port replaces the R register as the address source (although the result of the ALU operation is still written to the R register). If an instruction specifies B_3 or B_7 as an offset register, the offset is read from the data port. The YDTY instruction passes the data at the data port directly to the address port without affecting any R registers.

Data can also be transferred internally between R, B, C, and I registers:

TRANSFER	*INSTRUCTION(S)*
R to B	RTB
B to R	BTR
R to C	RTC
B to C	BTR, then RTC
I to R	ITR
I to B	ITR, then RTB
I to C	ITR, then RTC

Data input is latched during clock HI; data output is driven while the clock is HI and disabled while the clock is LO. This timing allows fast single-cycle data transfers. The application note *Optimize Data Transfers Between Word-Slice Components* describes various data transfer techniques. The data output delay can be modified by changing the instruction setup time, for a certain range of setup times; see *Sliding Window Timing*, below.

10.4 ADDRESS OUTPUT MODES

The ADSP-1410 can output addresses in one of two modes: latched or transparent. In the latched mode (the normal operating mode) the address port is transparent during clock HI and latched during clock LO. In the transparent mode, the address port is transparent for both phases of the clock, and the address output is referenced to the valid instruction input, rather than to the clock. Transparent mode provides the shortest possible address output delay, but because no latching is performed, the address outputs will change in response to changes at the instruction port.

In latched mode, the address output delay can be modified by changing the instruction setup time, for a certain range of setup times; see *Sliding Window Timing*, below.

In the transparent mode, if the clock is inactive (LO), the ADSP-1410 operates completely asynchronously. Because updated addresses are written to R registers on the falling edge of the clock, stopping the clock also prevents alteration of the R registers. This effect is beneficial when you want to use the ADSP-1410 as an ALU without changing any register states. If you restart the clock, you should ensure that a NOP is executed in the first cycle to prevent any inadvertent register writes.

10.5 SLIDING WINDOW TIMING

Sliding window timing pertains to the relationship between instruction setup time and address or data output delay time (latched mode only for address delay). The maximum output delay occurs when the minimum instruction setup time is provided. The minimum output delay occurs when the maximum usable instruction setup time is provided. If the instruction setup time is less than the maximum usable setup and greater than the minimum required setup, then the output delay and the instruction setup time sum to a constant. This constant is a "sliding window" about the rising clock edge, as shown in Figure 10.2. The sliding window provides flexibility in adjusting instruction setup and output delay to suit the needs of the system; you can reduce the actual output delay by increasing the instruction setup time, up to the maximum usable setup time, or you can reduce the necessary instruction setup time by tolerating a greater output delay. The maximum actual address output delay is determined as a function of the instruction setup time as follows:

10 System Interface

t_{LA} (max) = t_{LAx} (max) + t_{IS} (min) – t_{IS} (actual)

t_{LA} (max) = maximum actual address output delay
t_{LAx} (max) = t_{LAn} (max) (pre-update mode) or t_{LAp} (max) (post-update mode) maximum address output delay with minimum t_{IS}
t_{IS} (min) = minimum instruction setup
t_{IS} (actual) = actual instruction setup

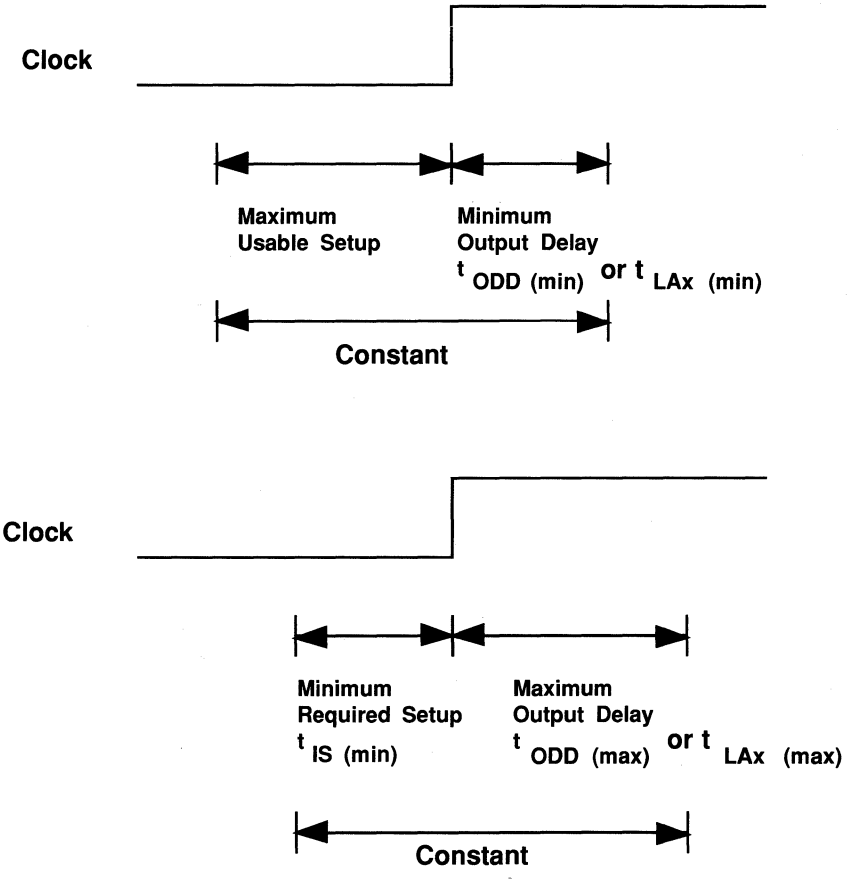


Figure 10.2 Sliding Window Timing

The maximum actual data output delay is determined as a function of the instruction setup time in the same manner:

t_{OD} (max) = t_{ODD} (max) + t_{IS} (min) – t_{IS} (actual)

t_{OD} (max) = maximum actual data output delay
t_{ODD} (max) = maximum data output delay with minimum t_{IS}
t_{IS} (min) = minimum instruction setup
t_{IS} (actual) = actual instruction setup

Refer to the most recent revision of the ADSP-1410 Data Sheet for the values of t_{LAx} (max), t_{ODD} (max) and t_{IS} (min).

10.6 COMPARE/ZERO

The Compare/Zero (CMP/Z) pin is a dual-function flag output that provides a way for external circuitry to monitor two internal status flags, the compare flag and the zero flag. These flags function as follows:

- The compare flag indicates the comparator status in a looping instruction. For looping instructions that increase the address value (YINC and YADD), the compare flag is HI if the address value is greater than or equal to the C register value. For looping instructions that decrease the address value (YDEC and YSUB), the compare flag is HI if the address value is less than or equal to the C register value.

- The zero flag indicates whether the result of a logical instruction is zero. If the result is zero, the zero flag is HI; if the result is not zero, the zero flag is LO. Logical instructions are YOR, YAND, and YXOR.

The CMP/Z output is LO when neither flag is active. It becomes valid during the cycle in which one of the instructions listed above is executed and remains active for one cycle.

10.7 BIT REVERSAL

Bit reversal transposes the bits of a word about its center. For the 16-bit address of the ADSP-1410, bit reversal interchanges the following bits:

$Y_{15} \leftrightarrow Y_0$
$Y_{14} \leftrightarrow Y_1$
$Y_{13} \leftrightarrow Y_2$
$Y_{12} \leftrightarrow Y_3$
$Y_{11} \leftrightarrow Y_4$
$Y_{10} \leftrightarrow Y_5$
$Y_9 \leftrightarrow Y_6$
$Y_8 \leftrightarrow Y_7$

10 System Interface

Bit-reversed addresses are used in implementations of algorithms such as a radix-2 fast Fourier transform (FFT). The addresses of the outputs of the FFT are the bit-reversed addresses of the corresponding inputs, as shown in Figure 10.3. An in-place implementation (output samples written back to input sample locations) of this FFT requires the generation of bit-reversed addresses.

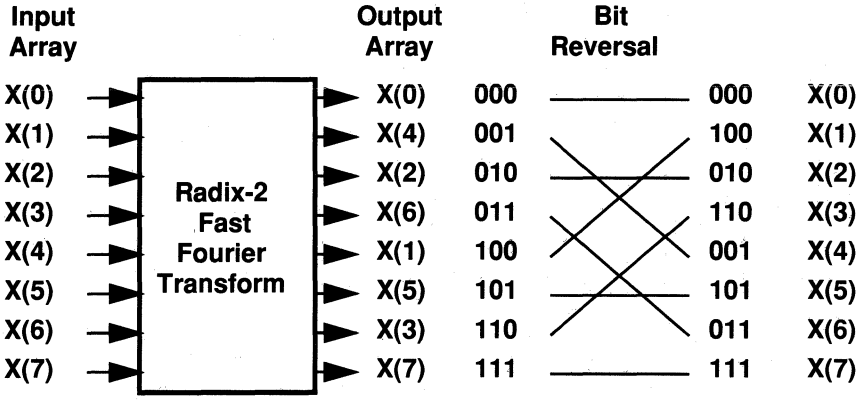


Figure 10.3 Output Addresses From FFT

The ADSP-1410 contains a hardware 16-bit reverser that is activated through the YREV instruction. YREV performs the following operations in the pre-update mode:

- Outputs the 16-bit-reversed version of the address in the specified R register
- Adds the offset in the specified B register to the original (non-bit-reversed) address
- Writes the result back to the R register

In the post-update mode, the address is bit-reversed and output after the offset is added to the original address. If the R register contains the starting address of an input sequence and the B register contains the address-spacing value (the number of address locations from one input sample to the next), successive executions of YREV produce 16-bit addresses in bit-reversed order.

If the input sequence contains fewer than 2^{16} locations, only the N bits required to address all of the input points need to be reversed. Figure 10.4 shows the bit reversal for N = 3 ($2^3 = 8$ data locations). You can generate this sequence, or any sequence of addresses whose N least significant bits are reversed, by loading the appropriate base address in the R register and offset value in the B register. The only restriction on this method is that the N least significant bits of the base address must be zeros; the 16–N most significant bits can be any combination of ones and zeros.

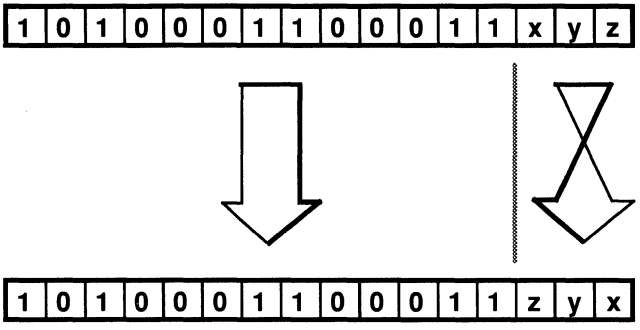


Figure 10.4 Three-Bit-Wide Address Reversal

As an example, suppose N = 3, providing enough addresses for eight input points if the addresses are consecutive. For this sequence, only the three least significant bits should be reversed, leaving the 13 most significant bits unchanged. The base address must have its three least significant bits set to zero; in this example, the base address is H#4C50, and the ending address is H#4C57.

The 3-bit reversal of the base address is the same as the unreversed base address, because of the requirement that the N least significant bits of the base address be zeros; N zeros reverses to N zeros. Thus, in this example, the first 3-bit-reversed address must be H#4C50. Because the ADSP-1410 performs only 16-bit reversals and not 3-bit reversals, you must store H#0A32, the 16-bit reversal of H#4C50, in the R register to get H#4C50 as the bit-reversed base address. When the YREV instruction is executed, H#4C50 is output, as shown in Figure 10.5.

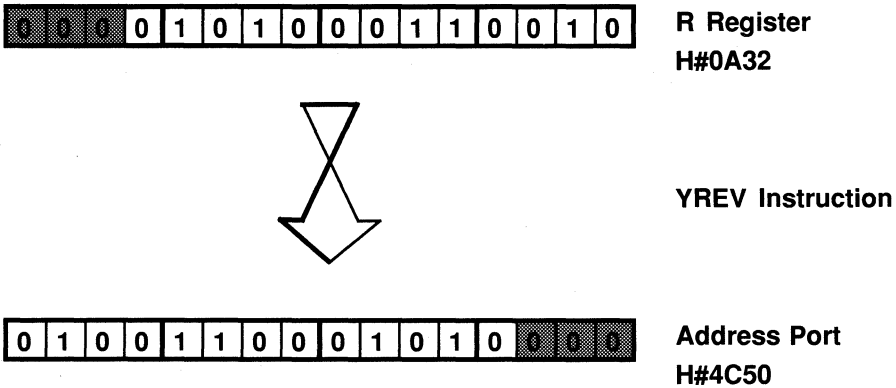


Figure 10.5 First Bit-Reversed Address

Note that in the R register value (H#0A32), the three bits that are being reversed are in the three most significant bit locations (bits 15, 14, and 13). When the 16-bit value is reversed, these three bits end up in the least significant bit locations (0, 1, and 2). To calculate the next bit-reversed address, you must add to the base address the address spacing value, which is 001 (binary) in this example. This value must be located in the three most significant bit positions in order to be added to the three bits that are to be reversed. The other 13 bits must be unchanged. Therefore, adding the value H#2000 (001 binary plus 13 least significant zeros) to the R register value will yield the correct address when the R register value is bit-reversed (see Figure 10.6). The B register stores the value H#2000, which is added to the R register by the YREV instruction. Subsequent executions of the YREV instruction yield the sequence of bit-reversed addresses.

In general, the offset to store in the B register is 2^{16-N}, where N is the number of bits being reversed. If the memory locations interleave real and imaginary data (every two locations contain the real part and imaginary part of a single data value), the offset should be $(2^{16-N}) \div 2$.

The application note *Variable Width Bit-Reversing with the ADSP-1410 Address Generator* gives examples of N-bit reversing for different address ranges within the 64K address space.

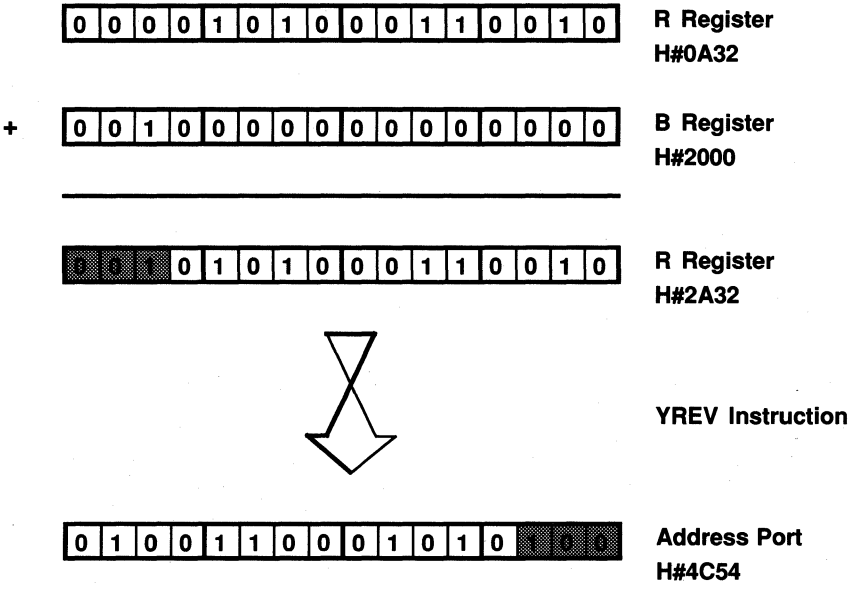


Figure 10.6 Second Bit-Reversed Address

10.8 RESET

Because the ADSP-1410 has no hardware reset, it can power up in any state. You cannot even guarantee it executes the instruction at its instruction port because it can power up with the AIR enabled. Therefore, there is a potential for data bus or address bus contention. You can prevent data bus contention by holding the clock input LO when you first apply power to the device; the data drivers are enabled only when the clock is HI.

The RST instruction takes precedence over all instructions, even if the AIR is enabled. RST clears all the bits in the control register and the instruction port and also places the address and data ports in the high impedance state; other registers are unaffected, as shown in Table 10.1. To ensure that RST is the first instruction executed, you should set up the RST instruction on the instruction port before starting the clock. This setup can be accomplished through a multiplexer that is enabled by the system reset signal. The same signal should gate the ADSP-1410 clock, as shown in Figure 10.7.

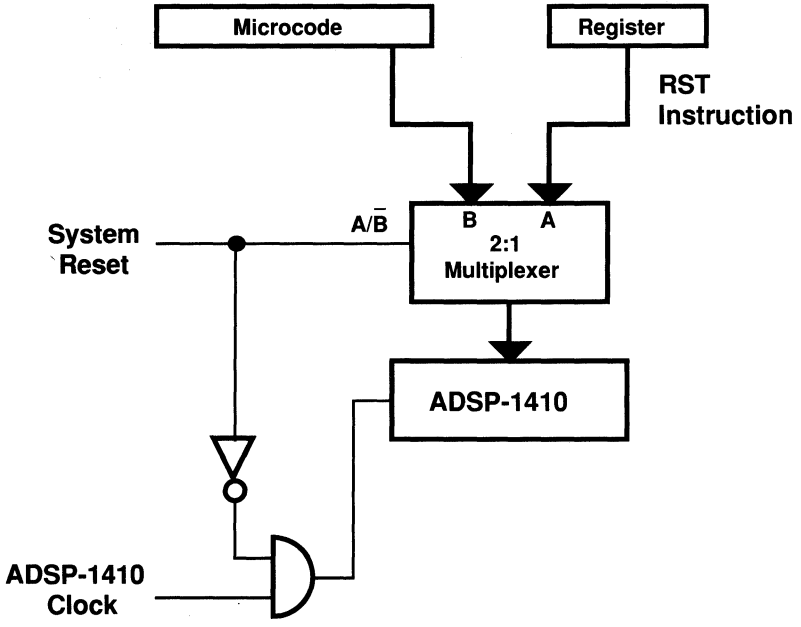


Figure 10.7 System Reset Initializes ADSP-1410

10 System Interface

LOCATION	*STATUS*
Address Registers	Unchanged
Offset Registers	Unchanged
Compare Registers	Unchanged
Initialization Registers	Unchanged
Control Register	All bits cleared
Alternate Instruction Register	Unchanged
Instruction Latch	All bits cleared
Address Port	High Impedance
Data Port	High Impedance

Table 10.1 Effect of RST Instruction

Instruction Set 11

11.1 INTRODUCTION

This chapter describes all ADSP-1410 instructions. The reference section describes each instruction in detail and provides one or more examples that use the instruction. This section is followed by a summary of the standard instruction mnemonics and opcodes. Both the reference section and summary are organized according to six functional groups and alphabetically by mnemonic within each group.

- Looping
- Register Transfer
- Logical and Shift
- Control Register
- Alternate Instruction Register
- Miscellaneous

The standard instruction mnemonics used in this manual relate directly to the instructions they represent. The mnemonics of all instructions that output an address begin with the letter Y. The mnemonics of instructions that transfer data from one port or register to another are composed of one or two letters representing the source, followed by the letter T, followed by one or two letters representing the destination. For example, BTR transfers data from a B register to an R register. The conventions used in this chapter are listed in Table 11.1.

11.2 INSTRUCTION REFERENCE

This section describes the operations performed by each ADSP-1410 instruction. Any values that are changed and any restrictions that apply are specified. The opcode for each instruction is listed, plus one or more short examples that illustrate usage. In the examples, user-selected bit values are underlined. Control register or control input values are indicated where relevant.

11 Instruction Set

Register and Port Name Abbreviations:

R_n	*nth* Address (R) register	$(0 \le n \le 15)$
B_m	*mth* Offset (B) register	$(0 \le m \le 7$; 3 or 7 = data port)
C_j	*jth* Comparison (C) register	$(0 \le j \le 3)$
I_j	*jth* Initialization (I) register	$(0 \le j \le 3)$
D	Data port (D_{15-0})	
D_x	Data bit *x*	$(0 \le x \le 15)$
Y	Address port (Y_{15-0})	
Y_x	Address bit *x*	$(0 \le x \le 15)$
CR_x	Control register bit *x*	$(0 \le x \le 10)$
AIR	Alternate Instruction Register	

Bit Codes:

rrrr	Four-bit R register index
rrr	Three-bit R register index (fourth bit in CR_7)
bb	Two-bit B register index (third bit in CR_8)
cc	Two-bit C register index
ii	Two-bit I register index
pp	Two-bit precision code
x	One-bit control register value

Functions:

ASR(x)	Arithmetic shift right function
LSL(x)	Logical shift left function
REV(x)	Bit reverse function

Table 11.1 Notation Terms

11.2.1 Looping

Looping instructions update or reinitialize an R register address based on the value of the address with respect to the value of the C register specified in the instruction opcode. They also output the R register value on the address port either before or after the update, depending on the update mode. These instructions are used frequently to implement circular buffers and modulo addressing.

The update operation differs for each looping instruction:

YADD	Add offset or reinitialize
YDEC	Decrement or reinitialize
YINC	Increment or reinitialize
YSUB	Subtract offset or reinitialize

Looping instructions output the address in a specified R register on the address port either before or after updating the address through an ALU operation, then compare the output address to a boundary value in a specified C register. If the compare status is true (the boundary value is reached or passed), the CMP/Z output goes HI to allow external monitoring. Internally, the compare status causes one of three actions to be performed, depending on the settings of control register bits 10 (conditional AIR execute) and 3-0 (initialization mask):

- If CR_{10} is set, the ADSP-1410 executes its next instruction from the AIR and ignores its instruction port.
- If CR_{10} is cleared and the bit in the initialization mask (CR_{3-0}) that corresponds to the C register is set, the R register is reinitialized with the value stored in the matching I register.
- If CR_{10} is cleared and the initialization bit is cleared, the comparator output has no internal effect. The R register is written with the output of the ALU operation.

If the compare status is not true (the boundary value is not reached or passed), the R register is written with the output of the ALU operation and the CMP/Z output remains LO.

If the DSEL input is asserted when a looping instruction is loaded, the address is supplied from the data port instead of the specified R register. The updated address is still written to the R register.

11.2.1.1 YADD

Output Address / Add Offset or Reinitialize

$Y \leftarrow R_n$ (Pre-Update Mode) or $Y \leftarrow R_n + B_m$ (Post-Update Mode)

IF ($Y \geq C_j$)
THEN $R_n \leftarrow I_j$
ELSE $R_n \leftarrow R_n + B_m$

Description:
Outputs the address in the specified R register on the address port either before (pre-update mode) or after (post-update mode) summing the address with the offset in the specified B register. If the output address is greater than or equal to the specified C register value, the R register is reinitialized with the value stored in the corresponding I register, provided the initialization mask bit in the control register is enabled and the conditional AIR execute mode is disabled ($CR_{10} = 0$). If the output address is less than the value in the specified C register, the R register is written with the sum of the address and offset value.

This instruction uses the R bank selection bit in the control register (CR_7) to select one of the two R register banks.

Opcode:

```
11 cc bb 1 rrr
   |  |     |
   |  |     +---- R register index (CR7 selects bank)
   |  +---------- B register index (CR8 selects bank)
   +------------- C/I register index
```

Examples:

```
11 00 01 1 011                    CR8 = 1, CR7 = 0
```

Outputs the address in R_3 either before or after adding the value in B_5 to the address. Compares the output address with the value in C_0. If the address is less than or equal to the C_0 value, the value in I_0 is written to R_3.

```
11 10 11 1 010                    CR8 = don't care, CR7 = 1
```

Outputs the address in R_{10} either before or after adding the value at the data port to the address. Compares the output address with the value in C_2. If the address is less than or equal to the C_2 value, the value in I_2 is written to R_{10}.

11 11 10 1 111 $CR_8 = 1$, $CR_7 = 1$, DSEL = 1

Reads the value at the data port and outputs this value on the address port either before or after adding to it the value in B_6. Compares the output address with the value in C_3. If the address is less than or equal to the C_3 value, the value in I_3 is written to R_{15}; otherwise, the sum of the data port value and the offset value is written to R_{15}.

11.2.1.2 YDEC

Output Address / Decrement or Reinitialize

$Y \leftarrow R_n$ (Pre-Update Mode) or $Y \leftarrow R_n - 1$ (Post-Update Mode)

IF $(Y \leq C_j)$
THEN $R_n \leftarrow I_j$
ELSE $R_n \leftarrow R_n - 1$

Description:
Outputs the address in the specified R register on the address port either before (pre-update mode) or after (post-update mode) decrementing the address by one. If the output address is less than or equal to the specified C register value, the R register is reinitialized with the value stored in the corresponding I register, provided the initialization mask bit in the control register is enabled and the conditional AIR execute mode is disabled ($CR_{10} = 0$). If the output address is greater than the value in the specified C register, the R register is written with the decremented address.

Opcode:

```
10 10 cc rrrr
      |  |
      |  └──── R register index
      └─────── C/I register index
```

Examples:

```
10 10 11 1110
```

Outputs and decrements the address in R_{14} and compares the output address with the value in C_3. If the address is less than or equal to the C_3 value, the value in I_3 is written to R_{14}.

```
10 10 01 0111                    DSEL = 1
```

Reads the value at the data port and outputs this value at the address port either before or after decrementing it. Compares the output address with the value in C_1. If the address is less than or equal to the C_1 value, the value in I_1 is written to R_7; otherwise, the data port value decremented by one is written to R_7.

11.2.1.3 YINC

Output Address / Increment or Reinitialize

$Y \leftarrow R_n$ (Pre-Update Mode) or $Y \leftarrow R_n + 1$ (Post-Update Mode)

IF $(Y \geq C_j)$
THEN $R_n \leftarrow I_j$
ELSE $R_n \leftarrow R_n + 1$

Description:
Outputs the address in the specified R register on the address port either before (pre-update mode) or after (post-update mode) incrementing the address by one. If the output address is greater than or equal to the specified C register value, the R register is reinitialized with the value stored in the corresponding I register, provided the initialization mask bit in the control register is enabled and the conditional AIR execute mode is disabled ($CR_{10} = 0$). If the output address is less than the value in the specified C register, the R register is written with the incremented address.

Opcode:

```
10 11 cc rrrr
      |  |
      |  +---- R register index
      +------- C/I register index
```

Examples:

```
10 11 10 0110
```

Outputs the address in R_6 either before or after incrementing it by one and compares the output address with the value in C_2. If the address is greater than or equal to the C_2 value, the value in I_2 is written to R_6.

```
10 11 00 1100
```

Outputs the address in R_{12} either before or after incrementing it by one and compares the output address with the value in C_0. If the address is greater than or equal to the C_0 value, the value in I_0 is written to R_{12}.

11.2.1.4 YSUB

Output Address / Subtract Offset or Reinitialize

$Y \leftarrow R_n$ (Pre-Update Mode) or $Y \leftarrow R_n - B_m$ (Post-Update Mode)

IF ($Y \leq C_j$)
THEN $R_n \leftarrow I_j$
ELSE $R_n \leftarrow R_n - B_m$

Description:
Outputs the address in the specified R register on the address port either before (pre-update mode) or after (post-update mode) subtracting the offset in the specified B register from the address. If the output address is less than or equal to the specified C register value, the R register is reinitialized with the value stored in the corresponding I register, provided the initialization mask bit in the control register is enabled and the conditional AIR execute mode is disabled ($CR_{10} = 0$). If the output address is greater than the value in the specified C register, the R register is written with the offset address.

This instruction uses the R bank selection bit in the control register (CR_7) to select one of the two R register banks.

Opcode:

```
11 cc bb 0 rrr
```

- rrr — R register index (CR_7 selects bank)
- bb — B register index (CR_8 selects bank)
- cc — C/I register index

Examples:

```
11 00 11 0 000                    CR8 = don't care, CR7 = 0
```

Outputs the address in R_0 either before or after subtracting the value at the data port (B_3 or B_7) from the address. Compares the output address with the value in C_0. If the address is less than or equal to the C_0 value, the value in I_0 is written to R_0; otherwise, the original value of R_0 less the data port value is written to R_0.

```
11 01 01 0 100                    CR8 = 1, CR7 = 1, DSEL = 1
```

Reads the value at the data port and outputs this value either before or after subtracting from it the value of B_5. Compares the output address with the value in C_1. If the address is less than or equal to the C_1 value, the value in I_1 is written to R_{12}; otherwise, the original data port value less the value of B_5 is written to R_{12}.

11.2.2 Register Transfer

Register transfer instructions are used to transfer data to and from internal registers:

BTD	Transfer from B to data port
BTR	Transfer from B to R
CTD	Transfer from C to data port
DTI	Transfer from data port to I
ITD	Transfer from I to data port
ITR	Transfer from I to R
RTD	Transfer from R to data port
YRTB	Output and transfer from R to B
YRTC	Output and transfer from R to C
YRTR	Output and transfer from R to same R

Register transfer instructions transfer values between the following places, as shown in Figure 11.1:

- *Between two internal registers.* A value can be transferred directly from any I or B register to any R register. The address in any R register can be output on the address port and written to a B or C register in one instruction. Transfers between two R registers can be accomplished in two instructions: one to transfer from the source R register to a B register, another to transfer from the B register to the destination R register.

- *Between an internal register and the data port.* Explicit instructions transfer data to the data port from R, B, C, and I registers. Transfers from the data port to an I register can be programmed explicitly. Transfers to the R, B, and C registers are accomplished by executing a transfer from an R register and asserting the DSEL pin, which substitutes the data port for the source R register. Both DSEL and the data input must meet input setup requirements.

To transfer values to and from the control register, see *Control Register Instructions* later in this chapter. To transfer values to and from the AIR, see *Alternate Instruction Register Instructions* later in this chapter.

11 Instruction Set

Instructions that begin with the letter Y also output the register value to the address port.

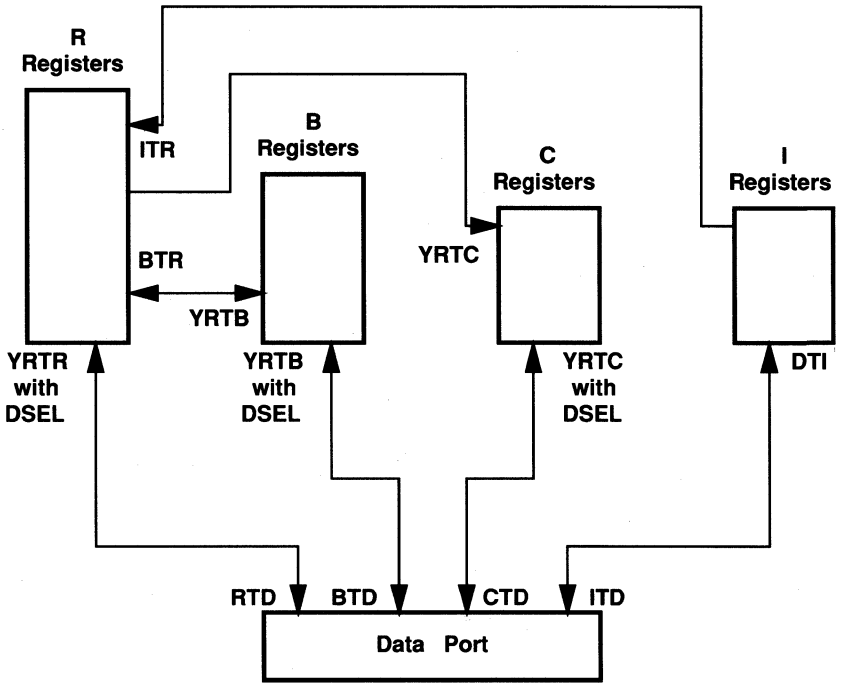


Figure 11.1 Data Transfer Paths

11.2.2.1 BTD

Transfer From B Register to Data Port

$D \leftarrow B_m$

Description:
Outputs the value in the specified B register on the data port. This instruction can be used to place a B register value on the data bus so that it can be used by other devices.

Opcode:

```
00 0011 01 bb
           └── B register index (CR8 selects bank)
```

Examples:

```
00 0011 01 10          CR8 = 1
```

Outputs the value of B_6 on the data port.

```
00 0011 01 00          CR8 = 0
```

Outputs the value of B_0 on the data port.

11.2.2.2 BTR

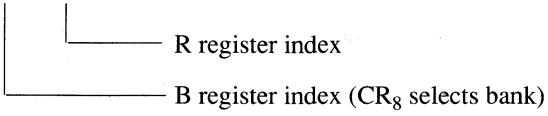
Transfer From B Register to R Register

$R_n \leftarrow B_m$

Description:
Loads the value in the specified B register into the specified R register. This instruction can be used to transfer an offset value to an R register for modification. The YRTB instruction returns the offset to the B register.

Opcode:

```
01 00 bb rrrr
```



Examples:

```
01 00 11 0110          CR8 = don't care
```

Loads the value at the data port (B_3 or B_7) into R_6.

```
01 00 01 1110          CR8 = 0
```

Loads the value of B_1 into R_{14}.

11.2.2.3 CTD

Transfer From C Register to Data Port

$D \leftarrow C_j$

Description:
Outputs the value in the specified C register on the data port. This instruction can be used to place the value of a C register on the data bus so that it can be used by other devices.

Opcode:

```
00 0011 00cc
          └──── C register index
```

Examples:

```
00 0011 00 10
```

Outputs the value of C_2 on the data port.

```
00 0011 00 00
```

Outputs the value of C_0 on the data port.

11.2.2.4 DTI

Transfer From Data Port to I Register

$I_j \leftarrow D$

Description:
Loads the value at the data port into the specified I register.

Opcode:
```
00 0011 11ii
           └──── I register index
```

Examples:
```
00 0011 11 01
```

Loads the value at the data port into I_1.

```
00 0011 11 10
```

Loads the value at the data port into I_2.

11.2.2.5 ITD

Transfer From I Register to Data Port

$D \leftarrow I_j$

Description:
Outputs the value in the specified I register on the data port. This instruction can be used by other devices to read I register values.

Opcode:

```
00 0011 10ii
          └──── I register index
```

Examples:

```
00 0011 10 00
```

Outputs the value in I_0 on the data port.

```
00 0011 10 11
```

Outputs the value in I_3 on the data port.

11.2.2.6 ITR

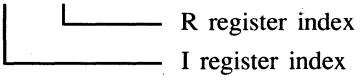
Transfer From I Register to R Register

$R_n \leftarrow I_j$

Description:
Loads the value in the specified I register into the specified R register. This instruction can be used to reinitialize an R register unconditionally.

Opcode:

```
10 00 ii rrrr
```



Examples:

```
10 00 01 0111
```

Loads the value in I_1 into R_7.

```
10 00 00 1010
```

Loads the value in I_0 into R_{10}.

11.2.2.7 RTD

Transfer From R Register to Data Port

$D \leftarrow R_n$

Description:
Outputs the address in the specified R register on the data port. This instruction can be used by other devices to read R register values. DSEL has no effect if asserted for this instruction.

Opcode:

```
00 0100 rrrr
        └──── R register index
```

Examples:

```
00 0100 1001
```

Outputs the value of R_9 on the data port.

```
00 0100 0101
```

Outputs the value of R_5 on the data port.

11.2.2.8 YRTB

Output Address and Transfer Address to B Register

$Y \leftarrow R_n$

$B_m \leftarrow R_n$

Description:
Outputs the address in the specified R register on the address port and writes the same address to the specified B register. If DSEL is asserted, the value at the data port is output to the address port and written to the specified B register:

$Y \leftarrow D$

$B_m \leftarrow D$

This feature can be used to load B registers from the data bus.

Opcode:

```
00 11 bb rrrr
      |  |
      |  +---- R register index
      +------- B register index (CR8 selects bank)
```

Examples:

```
00 11 01 0001                          CR8 = 1, DSEL = 0
```

Outputs the address in R_1 on the address port and writes the same address to B_5.

```
00 11 00 0001                          CR8 = 0, DSEL = 1
```

Reads the value at the data port and outputs this value on the address port; writes the same address to B_0. R_1 is unaffected.

11.2.2.9 YRTC

Output Address and Transfer Address to C Register

$Y \leftarrow R_n$

$C_j \leftarrow R_n$

Description:

Outputs the address in the specified R register on the address port and writes the same address to the specified C register. If DSEL is asserted, the value at the data port is output to the address port and written to the specified C register:

$Y \leftarrow D$

$C_j \leftarrow D$

This feature can be used to load C registers from the data bus.

Opcode:

```
00 10 cc rrrr
```

R register index

C register index

Examples:

```
00 10 11 0010                    DSEL = 0
```

Outputs the address in R_2 on the address port and writes the same value to C_3.

```
00 10 11 1010                    DSEL = 1
```

Reads the value at the data port and outputs this value on the address port; writes the same value to C_3. R_{10} is unaffected.

11.2.2.10 YRTR

Output Address and Transfer Address to Same Register

$Y \leftarrow R_n$

$R_n \leftarrow R_n$

Description:
Outputs the address in the specified R register on the address port. If DSEL is asserted, the value at the data port is output to the address port and written to the specified R register:

$Y \leftarrow D$

$R_n \leftarrow D$

This feature can be used to load R registers from the data bus.

Opcode:

```
00 0101 rrrr
        └──── R register index
```

Examples:

```
00 0101 1100                          DSEL = 0
```

Outputs the address in R_{12} on the address port.

```
00 0101 0101                          DSEL = 1
```

Reads the value at the data port, outputs this value on the address port, and writes the value to R_5.

11.2.3 Logical and Shift

Logical and shift instructions update the value of an R register by performing either a logical operation or shift operation on the existing value. They also output the R register on the address port either before or after the update operation, depending on the update mode.

YAND	Output and AND with B
YASR	Output and arithmetic shift right one bit
YLSL	Output and logical shift left one bit
YOR	Output and OR with B
YXOR	Output and XOR with B

Logical operations are AND, OR, and XOR; the first operand is the R register value, and the second operand is loaded from the specified B register. The left shift operation shifts the R register value one bit left logically (fills the vacated bit with a zero). The right shift operation shifts the R register value one bit right arithmetically (fills the vacated bit with a copy of the original most significant bit).

If the DSEL input is asserted when a logical instruction or shift instruction is loaded, the address is supplied from the data port instead of the specified R register. The updated value is still written to the R register.

The CMP/Z pin goes HI if the result of any logical instruction is zero. This output lets you use the zero result to condition some other event.

11 LOGICAL AND SHIFT YAND

11.2.3.1 YAND

Output Address and Logical AND With B Register

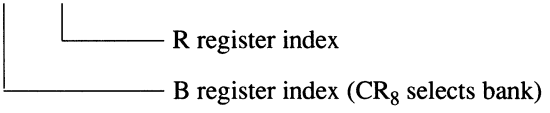
$Y \leftarrow R_n$ (Pre-Update Mode) or $Y \leftarrow (R_n \text{ AND } B_m)$ (Post-Update Mode)

$R_n \leftarrow (R_n \text{ AND } B_m)$

Description:
Outputs the address in the specified R register on the address port either before (pre-update mode) or after (post-update mode) ANDing the address with the value in the specified B register. Writes the AND result to the same R register.

Opcode:

```
01 10 bb rrrr
```



Examples:

```
01 10 00 1101                          CR8 = 1
```

Outputs the address in R_{13} either before or after ANDing it with the value in B_4. Writes the AND result to R_{13}.

```
01 10 11 1001                          CR8 = don't care
```

Outputs the address in R_9 either before or after ANDing it with the value read at the data port. Writes the AND result to R_9.

11.2.3.2 YASR

Output Address and Right Shift (Arithmetic)

$Y \leftarrow R_n$ (Pre-Update Mode) or $Y \leftarrow ASR(R_n)$ (Post-Update Mode)

$R_n \leftarrow ASR(R_n)$

Description:
Outputs the address in the specified R register on the address port either before (pre-update mode) or after (post-update mode) shifting the address one bit to the right. Writes the shifted result to the same R register. The shift is arithmetic; the vacated most significant bit is filled with a copy of the original most significant bit (the sign bit).

Opcode:

```
00 0111 rrrr
        └──── R register index
```

Example:

```
00 0111 1011
```

Outputs the address in R_{11} either before or after shifting the address right one bit. Writes the shifted result to R_{11}. For example, if the original value is

```
1000 1010 0011 0010
```

the shifted result is

```
1100 0101 0001 1001
```

11.2.3.3 YLSL

Output Address and Logical Left Shift

$Y \leftarrow R_n$ (Pre-Update Mode) or $Y \leftarrow LSL(R_n)$ (Post-Update Mode)

$R_n \leftarrow LSL(R_n)$

Description:
Outputs the address in the specified R register on the address port either before (pre-update mode) or after (post-update mode) shifting the address one bit to the left. Writes the shifted result to the same R register. The shift is logical; the vacated least significant bit is filled with a zero.

Opcode:

```
00 0110 rrrr
        └──── R register index
```

Example:

```
00 0110 1101
```

Outputs the address in R_{13} either before or after shifting the value left one bit. Writes the shifted result to R_{13}. For example, if the original value is

```
1000 1010 0011 0010
```

the shifted result is

```
0001 0100 0110 0100
```

11.2.3.4 YOR

Output Address and Logical OR With B Register

$Y \leftarrow R_n$ (Pre-Update Mode) or $Y \leftarrow (R_n \text{ OR } B_m)$ (Post-Update Mode)

$R_n \leftarrow (R_n \text{ OR } B_m)$

Description:
Outputs the address in the specified R register on the address port either before (pre-update mode) or after (post-update mode) ORing the address with the value in the specified B register. Writes the OR result to the same R register.

Opcode:

```
01 11 bb rrrr
      |  |
      |  +------ R register index
      |
      +--------- B register index (CR8 selects bank)
```

Examples:

```
01 11 10 0101          CR8 = 0
```

Outputs the address in R_5 either before or after ORing it with the value in B_2. Writes the OR result to R_5.

```
01 11 10 1101          CR8 = 1, DSEL = 1
```

Reads the value at the data port and outputs this value either before or after ORing it with the value in B_6. Writes the OR result to R_{13}.

11.2.3.5 YXOR

Output Address and Logical XOR With B Register

$Y \leftarrow R_n$ (Pre-Update Mode) or $Y \leftarrow (R_n \text{ XOR } B_m)$(Post-Update Mode)

$R_n \leftarrow (R_n \text{ XOR } B_m)$

Description:
Outputs the address in the specified R register on the address port either before (pre-update mode) or after (post-update mode) XORing the address with the value in the specified B register. Writes the XOR result to the same R register.

Opcode:

```
01 01 bb rrrr
      |  |
      |  +------ R register index
      |
      +--------- B register index (CR8 selects bank)
```

Examples:

```
01 01 11 1000                          CR8 = 1
```

Outputs the address in R_8 either before or after XORing it with the data port (B_7) value. Writes the XOR result to R_8.

```
01 01 00 1011                          CR8 = 1, DSEL = 1
```

Reads the value at the data port and outputs this value either before or after XORing it with the value of B_4. Writes the XOR result to R_{11}.

11.2.4 Control Register

Control register instructions write one or all bits of the control register with the specified value(s) or transfer the control register contents to the data port:

CRTD	Transfer from CR to data port
DTCR	Transfer from data port to CR
RST	Reset CR all bits
SELB	Set value of CR_8 (B bank select)
SELR	Set value of CR_7 (R bank select)
SETA	Set value of CR_{10} (Conditional AIR)
SETI	Set value of CR_{3-0} (Initialization mask)
SETP	Set value of CR_{5-4} (Precision mode)
SETU	Set value of CR_9 (Pre-/Post-update mode)
SETY	Set value of CR_6 (Transparent/Latched mode)

Control register bits are defined as follows:

Bit	*HI (1)*	*LO (0)*
0	I_0 Enabled	I_0 Disabled
1	I_1 Enabled	I_1 Disabled
2	I_2 Enabled	I_2 Disabled
3	I_3 Enabled	I_3 Disabled
5-4	Precision Mode Selection	

CR_5	CR_4	
0	0	One Chip, Single Precision
0	1	Two Chips, Double Precision, LSW
1	0	Two Chips, Double Precision, MSW
1	1	One Chip, Double Precision

Bit	*HI (1)*	*LO (0)*
6	Transparent Output Mode Selected	Latched Output Mode Selected
7	Upper R Register Bank Selected	Lower R Register Bank Selected
8	Upper B Register Bank Selected	Lower B Register Bank Selected
9	Post-Update Mode Selected	Pre-Update Mode Selected
10	Conditional AIR Execute Enabled	Conditional AIR Execute Disabled

11.2.4.1 CRTD

Transfer from Control Register to Data Port

$D_{10\text{-}0} \leftarrow CR_{10\text{-}0}$

Description:
Outputs the value in the control register ($CR_{10\text{-}0}$) on the data port ($D_{10\text{-}0}$).

Opcode:
`00 0010 1111`

Example:
If the control register bits are set as follows:

$CR_{3\text{-}0}$	All I registers enabled
$CR_{5\text{-}4}$	One-chip/double-precision mode
CR_6	Latched output mode
CR_7	Lower R register bank selected
CR_8	Upper B register bank selected
CR_9	Pre-Update mode
CR_{10}	Conditional AIR execution disabled

then the value output on the data port ($D_{10\text{-}0}$) is

`001 0011 1111`

and the values of $D_{15\text{-}11}$ are undefined.

11.2.4.2 DTCR

Transfer from Data Port to Control Register

$CR_{10\text{-}0} \leftarrow D_{10\text{-}0}$

Description:
Loads the 11 least significant bits of the data port ($D_{10\text{-}0}$) into the control register ($CR_{10\text{-}0}$).

Opcode:

```
00 0010 1110
```

Example:
If the value at $D_{10\text{-}0}$ is

```
010 1001 1010
```

the following modes are selected ($D_{15\text{-}11}$ are ignored):

$CR_{3\text{-}0}$	I_1 and I_3 enabled, I_0 and I_2 disabled
$CR_{5\text{-}4}$	Two-chip/double-precision mode, LS chip
CR_6	Latched output mode
CR_7	Upper R register bank selected
CR_8	Lower B register bank selected
CR_9	Post-Update mode
CR_{10}	Conditional AIR execution disabled

11.2.4.3 RST

Reset Control Register

$CR_{10\text{-}0} \leftarrow 0$

Description:
Resets all bits of the control register ($CR_{10\text{-}0}$). The RST instruction is always executed when it appears at the instruction port. It takes precedence even over an instruction enabled from the AIR. The RST instruction itself cannot be executed from the AIR; if the AIR is enabled when it contains the RST instruction, a NOP is performed.

Opcode:
`00 0000 0001`

Example:
Sets all CR bits to zero, selecting the following modes:

$CR_{3\text{-}0}$	All I registers disabled
$CR_{5\text{-}4}$	Single-precision mode
CR_6	Latched output mode
CR_7	Lower R register bank selected
CR_8	Lower B register bank selected
CR_9	Pre-Update mode
CR_{10}	Conditional AIR execution disabled

11.2.4.4 SELB

Select Upper or Lower B Register Bank

$CR_8 \leftarrow x$

Description:
Writes the bit value specified in the opcode to CR_8, selecting one of the two banks of four B registers to be used in all subsequent instructions that specify a B register. If CR_8 is set, the upper bank is selected; if CR_8 is cleared, the lower bank is selected.

Because either B_3 or B_7 represents the data port, the value of CR_8 is unimportant when one of these registers is specified.

Opcode:

```
00 0001 100 x
            └──── CR bit value
```

Examples:

```
00 0001 100 1
```

Selects the upper four B registers by setting CR_8 to a 1.

```
00 0001 100 0
```

Selects the lower four B registers by setting CR_8 to a 0.

11.2.4.5 SELR

Select Upper or Lower R Register Bank

$CR_7 \leftarrow x$

Description:
Writes the bit value specified in the opcode to CR_7, selecting one of the two banks of eight R registers to be used in all subsequent YADD and YSUB instructions. If CR_7 is set, the upper bank is selected; if CR_7 is cleared, the lower bank is selected.

Opcode:

```
00 0001 101 x
            └──── CR bit value
```

Examples:

```
00 0001 101 0
```

Selects the lower eight R registers by setting CR_7 to a 0.

```
00 0001 101 1
```

Selects the upper eight R registers by setting CR_7 to a 1.

11.2.4.6 SETA

Set or Clear Conditional AIR Execution

$CR_{10} \leftarrow x$

Description:
Writes the bit value specified in the opcode to CR_{10}, enabling or disabling conditional AIR execution. If CR_{10} is set, a true compare status in a looping instruction causes the instruction in the AIR to be executed on the next cycle. In this mode, reinitialization with I registers is disabled. If CR_{10} is cleared, the AIR can only be enabled by the AIRE input, and reinitialization with I registers is enabled according to the values of $CR_{3\text{-}0}$.

Opcode:

```
00 0001 010 x
            └──── CR bit value
```

Examples:

```
00 0001 010 1
```

Enables conditional AIR execution by setting CR_{10} to a 1.

```
00 0001 010 0
```

Disables conditional AIR execution by setting CR_{10} to a 0.

11.2.4.7 SETI

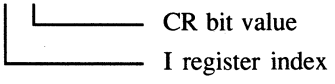
Set or Clear Initialization Mask Bit

$CR_{ii} \leftarrow x$

Description:
Writes the bit value specified in the opcode to one of the initialization mask bits (CR_{3-0}). The bit number is specified by two bits in the instruction opcode, *ii*. The mask bit controls reinitialization with the I register of the same number. Setting the bit enables a looping instruction to reinitialize an R register with the value in the I register. Clearing the bit disables reinitialization with the I register.

Opcode:

```
00 0010 0 ii x
```



Examples:

```
00 0010 0 11 1
```

Sets CR_3 to a 1, enabling I_3 for reinitialization.

```
00 0010 0 10 0
```

Clears CR_2, disabling I_2 for reinitialization.

11.2.4.8 SETP

Set Precision Mode

$CR_{5-4} \leftarrow pp$

Description:
Writes the two-bit value specified in the opcode to CR_{5-4}, setting the precision mode. Bit values correspond to precision modes as follows:

CR_5	CR_4	
0	0	single-precision mode
0	1	double-precision mode, LS chip
1	0	double-precision mode, MS chip
1	1	double-precision mode, one chip

To set an ADSP-1410 to the double-precision mode as the LS chip, you load this instruction specifying the bit values 01 for CR_{5-4}. If you assert the DSEL input while loading this instruction, the bit values are reversed to 10, causing the device to be set to the double-precision mode as the MS chip. This feature allows you to set the precision modes of both chips in a two-chip/double-precision configuration with the same instruction by asserting DSEL on the MS chip.

Opcode:

```
00 0010 10 pp
           └──── Precision code
```

Examples:

```
00 0010 10 11
```

Sets the one-chip/double-precision mode by setting CR_{5-4} to 11.

```
00 0010 10 01
```

Sets the two-chip/double-precision mode, LS chip, by setting CR_{5-4} to 01.

11.2.4.9 SETU

Set Update Mode

$CR_9 \leftarrow x$

Description:
Writes the bit value specified in the opcode to CR_9, setting the update mode. The update mode affects all instructions that output and update the address in an R register. If CR_9 is set, the address is output after being updated (post-update mode); if CR_9 is cleared, the address is output before being updated (pre-update mode).

Opcode:

```
00 0001 011 x
            └──── CR bit value
```

Examples:

```
00 0001 011 1
```

Sets the post-update mode by setting CR_9 to a 1.

```
00 0001 011 0
```

Sets the pre-update mode by setting CR_9 to a 0.

11.2.4.10 SETY

Set Address Output Mode to Transparent or Latched

$CR_6 \leftarrow x$

Description:
Writes the bit value specified in the opcode to CR_6, setting the address output mode. If CR_6 is set, the address port is always transparent (transparent mode). If CR_6 is cleared, the address port is transparent during clock HI and latched during clock LO (latched mode).

Opcode:

```
00 0001 001 x
            └──── CR bit value
```

Examples:

```
00 0001 001 1
```

Sets the transparent address output mode by setting CR_6 to a 1.

```
00 0001 001 0
```

Sets the latched address output mode by setting CR_6 to a 0.

11.2.5 Alternate Instruction Register

Alternate Instruction Register (AIR) instructions transfer data to and from the AIR. The instruction in the AIR is enabled for execution by the AIRE input unconditionally or by a true comparator status in the conditional AIR execute mode.

The AIR can be loaded from either the data port or the instruction port. Loading from the instruction port requires two cycles; DSEL must be asserted for the first cycle. This requirement allows you to load the AIRs of several ADSP-1410 devices that share the same instruction bus selectively.

LDA	Transfer from instruction port to AIR
RDA	Transfer from AIR to data port
WRA	Transfer from data port to AIR

11.2.5.1 LDA

Transfer from Instruction Port to AIR on Next Cycle

$AIR \leftarrow I_{9-0}$

Description:
Causes the instruction that appears at the instruction port on the next cycle to be loaded into the AIR, not executed; a NOP is performed instead. The DSEL input must be asserted along with the LDA instruction in order for AIR loading to take place. This feature is useful if you want to load only some of several ADSP-1410 devices that share the same instruction bus. The LDA instruction can be presented to all devices, but only the devices for which DSEL is asserted will load their AIRs on the next cycle.

You should not attempt to load the RST instruction into the AIR. If RST appears at the instruction port on the cycle following the LDA instruction, it is executed, not loaded into the AIR.

Opcode:
```
00 0001 1110
```

11.2.5.2 RDA

Transfer from AIR to Data Port

$D_{9\text{-}0} \leftarrow$ AIR

Description:
Outputs the value in the AIR on the ten least significant data bits ($D_{9\text{-}0}$). Bits $D_{15\text{-}10}$ are undefined.

Opcode:
`00 0010 1101`

11.2.5.3 WRA

Transfer from Data Port to AIR

$AIR \leftarrow D_{9\text{-}0}$

Description:
Loads the value on the ten least significant data bits ($D_{9\text{-}0}$) into the AIR.

Opcode:
```
00 0010 1100
```

11.2.6 Miscellaneous

Miscellaneous instructions perform the following operations:

NOP	No operation
YDTY	Transfer from data port to address port
YREV	Output address in bit-reversed format and add offset

11.2.6.1 NOP

No Operation

Description:
Performs no operation. All internal conditions are preserved. The address port and data port are placed in the high-impedance state.

Opcode:
00 0000 0000

11.2.6.2 YDTY

Transfer from Data Port to Address Port

$Y \leftarrow D$

Description:
The value at the data port ($D_{15\text{-}0}$) is read and output directly to the address port ($Y_{15\text{-}0}$). R registers are not affected.

Opcode:
`00 0001 1111`

11.2.6.3 YREV

Output Address in Bit-Reversed Format and Add Offset

$Y \leftarrow REV(R_n)$ (Pre-Update Mode)

or $Y \leftarrow REV(R_n + B_m)$ (Post-Update Mode)

$R_n \leftarrow R_n + B_m$

Description:
Bit-reverses and outputs the address in the specified R register either before (pre-update mode) or after (post-update mode) summing it with the offset in the specified B register. The non-reversed sum of the original value and the offset value is written to the same R register unconditionally.

Opcode:

```
10 01 bb rrrr
      |  |
      |  +------ R register index
      +--------- B register index (CR8 selects bank)
```

Examples:

```
10 01 01 0101                          CR8 = 1
```

Outputs a bit-reversed version of the address in R_5 either before or after adding the offset in B_5 to the original address. Writes the non-reversed sum of the original address and the offset value to R_5.

```
10 01 00 1000                          CR8 = 1, DSEL = 1
```

Outputs a bit-reversed version of the value read at the data port either before or after adding to it the offset in B_4. Writes the non-reversed sum of the original data port value and the offset value to R_8.

11 Instruction Set

11.3 MNEMONIC AND OPCODE SUMMARY

This section provides a quick reference to the ADSP-1410 instruction mnemonics and opcodes. The instructions are divided into six groups according to function. Within each group, the instructions are listed alphabetically by the standard mnemonic names used throughout this manual. The following bit codes are used:

rrrr	Four-bit R register index
rrr	Three-bit R register index (fourth bit in CR_7)
bb	Two-bit B register index (third bit in CR_8)
cc	Two-bit C register index
ii	Two-bit I register index
pp	Two-bit precision code
x	One-bit control register value

Control register bits are defined as follows:

Bit	*HI (1)*	*LO (0)*
0	I_0 Enabled	I_0 Disabled
1	I_1 Enabled	I_1 Disabled
2	I_2 Enabled	I_2 Disabled
3	I_3 Enabled	I_3 Disabled
5-4	Precision Mode Selection	

CR_5	CR_4	
0	0	One Chip, Single Precision
0	1	Two Chips, Double Precision, LSW
1	0	Two Chips, Double Precision, MSW
1	1	One Chip, Double Precision

Bit	*HI (1)*	*LO (0)*
6	Transparent Output Mode Selected	Latched Output Mode Selected
7	Upper R Register Bank Selected	Lower R Register Bank Selected
8	Upper B Register Bank Selected	Lower B Register Bank Selected
9	Post-Update Mode Selected	Pre-Update Mode Selected
10	Conditional AIR Execute Enabled	Conditional AIR Execute Disabled

Looping

YADD	`11 ccbb 1rrr`	Output Address and Add Offset or Reinitialize
YDEC	`10 10cc rrrr`	Output Address and Decrement or Reinitialize
YINC	`10 11cc rrrr`	Output Address and Increment or Reinitialize
YSUB	`11 ccbb 0rrr`	Output Address and Subtract Offset or Reinitialize

Instruction Set 11

Register Transfer

BTD	00 0011 01bb	Transfer From B Register To Data Port
BTR	01 00bb rrrr	Transfer From B Register To R Register
CTD	00 0011 00cc	Transfer From C Register To Data Port
DTI	00 0011 11ii	Transfer From Data Port To I Register
ITD	00 0011 10ii	Transfer From I Register To Data Port
ITR	10 00ii rrrr	Transfer From I Register To R Register
RTD	00 0100 rrrr	Transfer From R Register To Data Port
YRTB	00 11bb rrrr	Output Address and Transfer To B Register
YRTC	00 10cc rrrr	Output Address and Transfer To C Register
YRTR	00 0101 rrrr	Output Address and Transfer To Same Register

Logical and Shift

YAND	01 10bb rrrr	Output Address and Logical AND With B Register
YASR	00 0111 rrrr	Output Address and Arithmetic Right Shift
YLSL	00 0110 rrrr	Output Address and Logical Left Shift
YOR	01 11bb rrrr	Output Address and Logical OR With B Register
YXOR	01 01bb rrrr	Output Address and Logical XOR With B Register

Control Register

CRTD	00 0010 1111	Transfer From Control Register To Data Port
DTCR	00 0010 1110	Transfer From Data Port To Control Register
RST	00 0000 0001	Reset Control Register
SELB	00 0001 100x	Select B Register Bank
SELR	00 0001 101x	Select R Register Bank
SETA	00 0001 010x	Enable/Disable Conditional AIR Execution
SETI	00 0010 0iix	Enable/Disable I Register
SETP	00 0010 10pp	Set Precision Mode
SETU	00 0001 011x	Set Update Mode
SETY	00 0001 001x	Set Address Output Mode

Alternate Instruction Register

LDA	00 0001 1110	Transfer From Instruction Port To AIR Next Cycle
RDA	00 0010 1101	Transfer From AIR To Data Port
WRA	00 0010 1100	Transfer From Data Port To AIR

Miscellaneous

NOP	00 0000 0000	No Operation
YDTY	00 0001 1111	Transfer From Data Port To Address Port
YREV	10 01bb rrrr	Output Address Bit-Reversed and Add Offset

Analog Devices
Digital Signal Processing Division
One Technology Way
P.O. Box 9106
Norwood, MA 02062-9106
(617) 329-4700

E1070-5-4/87