UNIX™ MICROSYSTEM

WE® 32100 MICROPROCESSOR INFORMATION MANUAL

MAXICOMPUTING IN MICROSPACE

AT&T

451-000

UNIX™ MICROSYSTEM

WE® 32100 MICROPROCESSOR INFORMATION MANUAL

451-000

ACKNOWLEDGEMENTS

Prepared and published by
Document Development Organization — Microelectronics Projects Group
AT&T Technologies, Inc., Morristown

for the

Microsystem Product Management
AT&T Technologies, Inc.

and the

4516 Microsystems Laboratory
AT&T Bell Laboratories, Holmdel

A WORD ABOUT TRADEMARKS . . .

The following trademarks are mentioned in this manual:

WE® 32100 Microprocessor

WE® 32101 Memory Management Unit

WE® 32102 Clock

WE® 321AP Microprocessor Analysis Pod

WE® 321DS Microprocessor Development System

WE® 321EB Microprocessor Evaluation Board

WE® 321SD Development Software Programs

WE® 321SE Software Evaluation Program

WE® 321SG Software Generation Programs

are registered trademarks of AT&T Technologies, Inc.

AT&T 3B20S Computers is a trademark of AT&T.

UNIX™ Operating System is a trademark of AT&T Bell Laboratories.

PDP™ 11/70 Computer and *VAX*™ 11/780 Computer are trademarks of Digital Equipment Corporation.

IBM® 370 Computer is a registered trademark of the IBM Corporation.

104287958
IM232100CPU0115

January 1985

WE® 32100 Microprocessor

Information Manual

The information contained herein is subject to change.

FOREWORD

This manual contains information on the *WE* 32100 Microprocessor that is essential to computer designers, software architects, and system design engineers. The support software and development tools available simplify system integration for this complex 32-bit microprocessor. This issue contains a description of the version SVR2.0 of the *WE* 321SG Software Generation Programs.

Additional information is available in the form of data sheets, application notes, and on-line documentation from the *UNIX* Operating System.

For additional information contact your Sales Account Representative or call:
□ Commercial sales: 1-800-372-2447
□ AT&T and Associated Company sales: (215) 770-3204 or (CORNET) 8+624-3204.

To obtain additional copies of this manual, Select Code 451-000, call:
□ 1-800-432-6600.

WE 32100 MICROPROCESSOR INFORMATION MANUAL

CONTENTS

CHAPTER 1. INTRODUCTION

CHAPTER 2. ARCHITECTURE AND BUS OPERATION

CHAPTER 3. INSTRUCTION SET AND ADDRESSING MODES

CHAPTER 4. OPERATING SYSTEM CONSIDERATIONS

CHAPTER 5. SOFTWARE GENERATION PROGRAMS

GLOSSARY AND ACRONYMS

INDEX

LIST OF FIGURES

LIST OF TABLES

Chapter 1

Introduction

CHAPTER 1. INTRODUCTION

CONTENTS

1. INTRODUCTION

This chapter introduces the *WE* 32100 Microprocessor and summarizes the support products available for it. The chapters describing the *WE* 32100 Microprocessor architecture, instruction set, operating system considerations, and software generation programs are also introduced.

1.1 Overview

The *WE* 32100 Microprocessor is a high-performance, single-chip, 32-bit central processing unit designed for efficient operation in a high-level language environment. The *WE* 32100 Microprocessor represents a state-of-the-art concept in microprocessor architecture, providing one of the most powerful and extensive instruction sets available with any microprocessor. The *WE* 32100 Microprocessor, packaged in a 132-pin ceramic pin array, is shown on Figure 1-1.

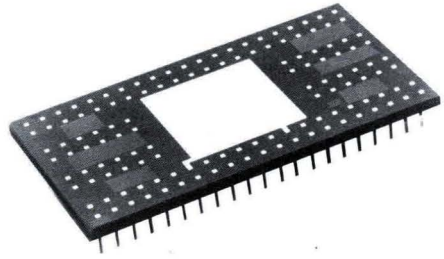


Figure 1-1. The *WE* 32100 Microprocessor

The system memory space is addressed over a full 32-bit address bus using either physical or virtual addresses. The 32-bit address bus produces a vast memory space of more than four billion bytes which increases the flexibility of memory organization and provides ample space for the storage of software and data. Data can be read or written over the separate 32-bit data bus in byte (8-bit), halfword (16-bit), or word (32-bit) lengths.

The *WE* 32100 Microprocessor is an efficient execution vehicle for operating systems and high-level languages. The operating system instructions included in the instruction set establish an environment that permits process switching and interrupt handling with a minimum of operating system support. Other instructions allow the use of coprocessors and provide the necessary signals for interfacing with the *WE* 32101 Memory Management Unit for virtual memory systems.

Software support for the *WE* 32100 Microprocessor is available through the *WE* 321SG Software Generation Programs (SGP). This collection of programs and utilities provides everything necessary for rapid development of software. The high-level development language is the C language, and the entire SGP resides in the *UNIX* Operating System. The SGP includes a C compiler, an assembler, a link editor, and various utility programs.

Development support is available through the *WE* 321DS Microprocessor Development System. The development system is a powerful development tool that can expedite the integration of hardware and software into a finished application. It permits the debugging of hardware and software to occur in parallel. The development system components include the *WE* 321AP Microprocessor Analysis Pod, the *WE* 321SD Development Software Programs, a *UNIX* System Host, and a logic analyzer. The modular design of the development system enables the user to configure the system for maximum productivity from initial hardware debug through the final stage of hardware and software integration. The *WE* 321AP Microprocessor Analysis Pod is shown on Figure 1-2.

Prototyping and performance evaluation support is available through the *WE* 321EB Microprocessor Evaluation Board. The evaluation board is a single-board microcomputer evaluation system that provides a prototyping vehicle to evaluate the hardware and software capabilities and performance of the *WE* 32100 Microprocessor in an application environment. The board is supplied with a *WE* 32100 Microprocessor as the CPU, a *WE* 32101 Memory Management Unit, a *WE* 32102 Clock, a ROM-based monitor, read/write memory (RAM), and sockets for additional memory. Also included are address decoding circuitry, RS-232C ports, programmable parallel I/O lines, programmable interval timers, and an interrupt controller. The *WE* 321EB Microprocessor Evaluation Board is shown on Figure 1-3.

1.2 Architecture

The *WE* 32100 Microprocessor performs all the system address generation, control, memory access, and processing functions required in a 32-bit microcomputer system. Execution speed is enhanced by its unique pipelined architecture. Using this architecture, the microprocessor overlaps the execution of instructions while tracking each separately. In addition, as each instruction is fetched from memory it is cached in an internal instruction cache, resulting in even greater operating efficiency.

The CPU utilizes a combination of address and data strobes and interface and control signals to provide the bus protocol required for efficient transfer of data. The protocol facilitates interfacing to commercial memories and peripherals, as well as providing wait-state generation for handshaking with slow peripherals. In addition, the CPU also provides special coprocessor signals for a high throughput coprocessing environment.

The architecture and a bus protocol for the *WE* 32100 Microprocessor is discussed in **Chapter 2. ARCHITECTURE AND BUS OPERATION.**

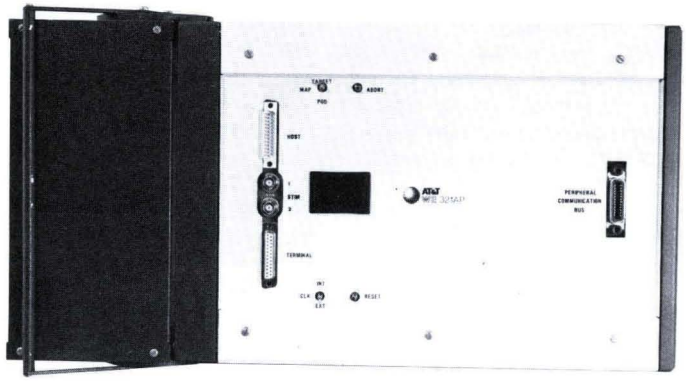


Figure 1-2. *WE* 321AP Microprocessor Analysis Pod

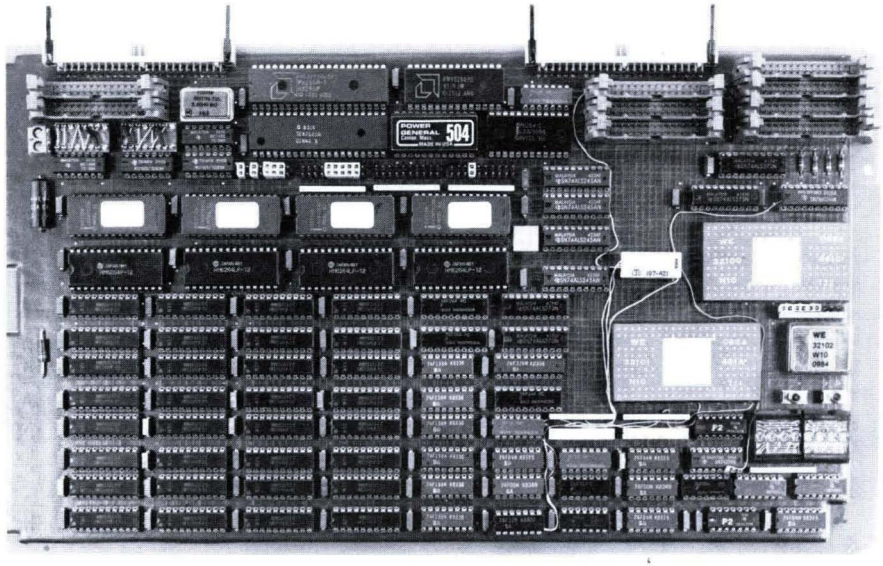


Figure 1-3. *WE* 321EB Microprocessor Evaluation Board

1.3 Instruction Set

The *WE* 32100 Microprocessor supports a powerful instruction set that includes standard data transfer, arithmetic, and logical operations for microprocessors, plus several unique operations. Its many program control instructions (branch, jump, return) provide flexibility for altering the sequence of execution. Other instructions are designed to aid in process switching for operating systems by manipulating the context of the processor with a minimum of code. In addition, special coprocessor instructions are included in the instruction set to implement a high-speed interface with special purpose coprocessors planned for the *WE* 32100 Microprocessor.

Eighteen addressing modes are provided that include special high-level language support modes such as frame pointer short offset and argument pointer short offset. These modes are designed for referring to local variables of high-level functions and function arguments.

Chapter 3. INSTRUCTION SET AND ADDRESSING MODES contains a detailed description of the *WE* 32100 Microprocessor Instruction Set.

1.4 Operating System Support

The *WE* 32100 Microprocessor is designed for high-level language and operating system support. To aid in the design of process-oriented systems, it provides:

- four execution privilege levels: kernel, executive, supervisor, and user
- flexible transfer of execution control between privilege levels
- capability to have the operating system contained within the address space of every process
- support of explicit process switching by a scheduler
- implicit switching of processes through the interrupt structure
- layered exception handling structure, with different mechanisms used for different exceptions.

The processor groups all of the switchable process context into a compact area in memory called the process control block. This feature, plus the use of the special operating system instructions and microsequences, provides the programmer with an excellent tool for the creation and support of process-oriented systems.

Chapter 4. OPERATING SYSTEM CONSIDERATIONS discusses the techniques for efficient operating system design using the *WE* 32100 Microprocessor and also describes the use of the *WE* 32101 Memory Management Unit in a virtual memory operating system.

1.5 Software Generation Programs

The *WE* 321SG Software Generation Programs (SGP) is a package of support tools used to create and test programs for the *WE* 32100 Microprocessor. The SGP runs under the *UNIX* Operating System and uses many features of the *UNIX* System shell. The SGP allows the programmer to generate code in the high-level C language and test programs at the source level. This improves productivity and program accuracy by freeing programmers from the details of the hardware architecture associated with assembly language programming.

The SGP contains a C compiler that converts C language programs into assembly language programs. The assembly language programs are ultimately translated into object files by the SGP assembler for the *WE* 32100 Microprocessor and link-edited into executable load modules by the link editor (also contained in the SGP). Each of these tools preserves all symbolic information necessary for meaningful symbolic testing at the source level. The SGP also provides a variety of utilities that read and manipulate object files.

The SGP is described in detail in **Chapter 5. SOFTWARE GENERATION PROGRAMS.**

Chapter 2

Architecture and Bus Operation

CHAPTER 2. ARCHITECTURE & BUS OPERATION

CONTENTS

CONTENTS

2. *WE* 32100 MICROPROCESSOR OVERVIEW

The *WE* 32100 Microprocessor is the first 32-bit microprocessor with separate 32-bit address and data buses. Using either physical or virtual addresses, the 32-bit address bus can access over four billion (2^{32}) bytes of system memory or peripherals. Data is read or written over the 32-bit bidirectional data bus in either byte (8-bit), halfword (16-bit), or word (32-bit) lengths and is processed internally over 32-bit internal data paths.

The execution speed of the microprocessor is enhanced by an internal instruction queue and an internal instruction cache that store prefetched instructions. Also, the microprocessor's extensive use of pipelining allows overlapping of the execution of instructions while tracking each one individually. Should a fault or interrupt occur during instruction execution, the instruction that caused it can be easily determined and execution restarted. This feature is essential for systems with demand-paged memory management.

Using a group of address and data strobes and interface and control signals, the microprocessor controls information flow over the address and data buses. These signals provide the timing required for transfer of data and facilitate interfacing to commercial memories and peripherals. The microprocessor also accommodates wait-state generation to allow handshaking with slow peripherals.

The *WE* 32100 Microprocessor consists of the four major sections shown on Figure 2-1. These are the main controller, the fetch unit, the execute unit, and the bus interface control. The main controller is responsible for acquiring and decoding instruction opcodes and directing the action of the fetch and execute controllers as the specified instruction is executed. The main controller also has the responsibility of responding to and directing the handling of interrupts and exception conditions.

The fetch unit handles the instruction stream and performs memory-based operand accesses. It consists of a fetch controller, an instruction cache, an instruction queue, an immediate and displacement extractor, and an address arithmetic unit (AAU). The fetch controller directs the action of the elements in the fetch unit. The instruction cache is a 64 by 32-bit on-chip cache which is used to increase the microprocessor's performance by reducing external memory reads for instruction fetches. When an instruction fetch from memory occurs, instruction data is placed in the cache and in the instruction queue. If that instruction data is needed again, it is fetched from the cache rather than from external memory, which improves performance. The instruction queue is an 8-byte first-in-first-out queue that stores prefetched instructions. Instructions are taken from the queue for execution, and the fetch controller fills it asynchronously with respect to instruction execution. The immediate and displacement extractor provides address calculation data to the AAU for its use in calculating 32-bit addresses.

The execute unit performs all arithmetic and logic operations, performs all shift and rotate operations, and computes condition flags. It consists of:

- an execute controller that directs the actions of the elements in the execute unit

- sixteen 32-bit registers that are user-accessible and include:
  - nine general-purpose registers (r0—r8)
  - seven dedicated registers (r9—r15)
- working registers that are used exclusively by the microprocessor and are not user-accessible
- a 33-bit ALU that performs arithmetic operations on 32-bit data, with an extra bit that is used whenever an operation requires a carry or borrow beyond 32 bits.

The bus interface control provides all the strobes and control signals necessary to implement the interface with peripherals.

The *WE* 32100 Microprocessor pin assignments are summarized in **2.17 Pin Assignments.**

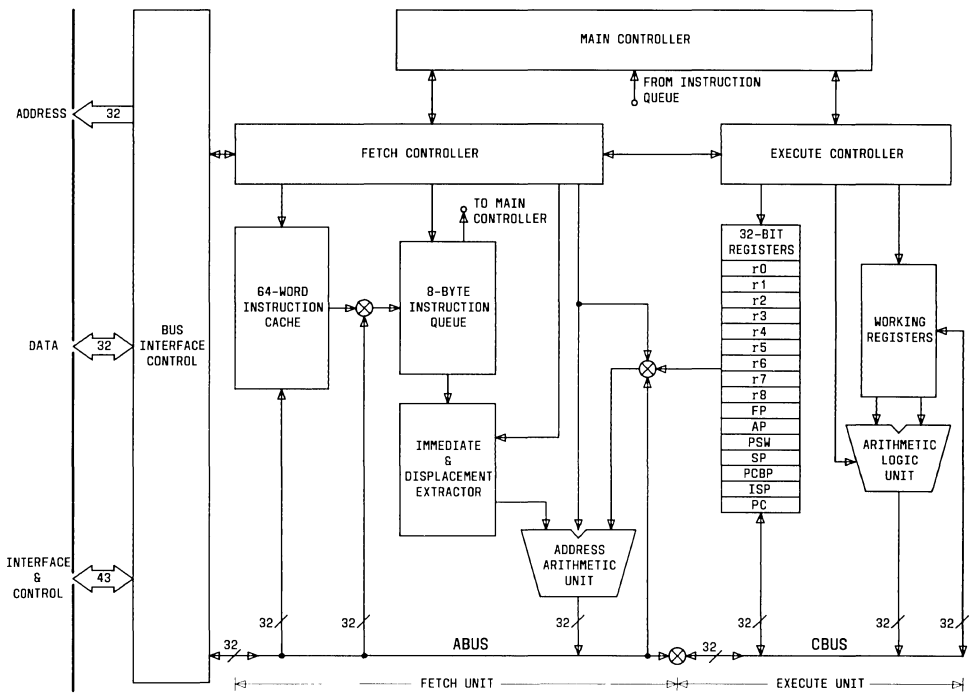


Figure 2-1. *WE* 32100 Microprocessor Block Diagram

2.1 USER REGISTERS

Figure 2-2 shows the programming model for the microprocessor's sixteen 32-bit registers (r0—r15). This register set is designed for efficient support of high-level language program execution. All of these registers, except for the program counter (r15) and the processor status word (r11), may be accessed in any addressing mode. The processor status word (r11), process control block pointer (r13), and interrupt stack pointer (r14) are privileged registers. These may be read at any time, but may be written only when the microprocessor is in kernel mode (i.e., the operating system is in control). The other registers may be read or written in any of the four execution levels.

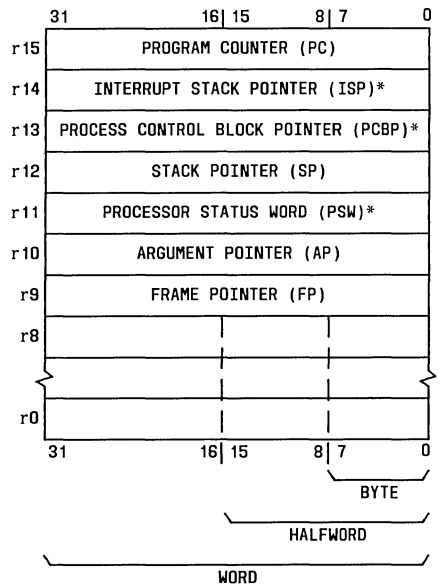


Figure 2-2. Programmer's Model for User Registers

2.1.1 General-Purpose Registers (r0—r8)

The nine general-purpose registers may be used for high-speed accumulation, for addressing, or for temporary data storage. The first three registers (r0—r2) are the microprocessor's *scratch* registers. These three registers are used by the C compiler to store temporary values during expression evaluation. They also pass and return specific values during procedure calls. For example, r0 should always be used to return the value of a procedure. If a floating point double value is returned from a procedure, it is stored in r0 and r1. If a procedure returns a structure, then the pointer to that structure should be returned to r2. In addition, registers r0—r2 are implicitly used by the data transfer instructions MOVBLW (move block of words), STRCPY (string copy), and STREND (string end) and also by the MVERNO (move version number), INTACK (interrupt acknowledge), ENBVJMP (enable virtual pin and jump), DISVJMP (disable virtual pin and jump), GATE (system-call), and CALLPS (call process) operating system instructions.

2.1.2 Frame Pointer

The frame pointer (FP), r9, points to the beginning location in the stack of a function's local variables. It is affected implicitly only by the save register (SAVE) and the restore register (RESTORE) instructions.

2.1.3 Argument Pointer

The argument pointer (AP), r10, points to the beginning location in the stack where a set of arguments for a function has been pushed. The AP is affected implicitly only by the procedure call (CALL) and return (RET) instructions.

2.1.4 Processor Status Word

The processor status word (PSW), r11, contains status information about the microprocessor and the current process. It is divided into 14 fields, as shown on Figure 2-3. Although the PSW is a privileged register, the microprocessor may alter some of its fields at any execution level. Most instructions alter the N, Z, V, and C bits (condition flags) in the PSW. In general, the PSW changes as a whole *only* when a process switch occurs. The final values of the PSW bits are based on the result of the last calculation and are latched into the PSW at the end of the instruction. The PSW may not be referenced in some addressing modes.

Table 2-1 contains a description of each of the processor status word fields.

Bit	31 26	25	24	23	22	21 18	17	16 13	12 11	10 9	8	7	3	2	1 0
Field	Unused	CFD	QIE	CD	OE	NZVC	TE	IPL	CM	PM	R	I	ISC	TM	ET

Figure 2-3. Processor Status Word

Table 2-1. Processor Status Word Fields

Bit(s)	Field	Contents	Description
0—1	ET	Exception Type	This read-only field indicates the type of exception generated during operations and is interpreted as: **Code** — **Description** Bit 1 Bit 0 0 0 On Reset Exception 0 1 On Process Exception 1 0 On Stack Exception 1 1 On Normal Exception (See **2.12 Exceptional Conditions.**)
2	TM	Trace Mask	The read-only TM field enables masking of a trace trap. This bit masks the trace enable (TE) bit for the duration of one instruction to avoid a trace trap. The TM bit is set (1) at the beginning of every instruction and cleared (0) as part of every microsequence that performs a context switch or a return from gate (RETG) or when any fault or interrupt is detected and responded to.
3—6	ISC	Internal State Code	This 4-bit code distinguishes between exceptions of the same exception type. The ISC is a read-only field. (See **2.15 Exceptional Conditions.**)
7—8	RI	Register-Initial Context	These bits control the context switching strategy. The I bit (bit 7) determines if a process executes from initial or intermediate saved context. The R bit (bit 8, read only) determines if the registers of a process should be saved during a process switch. It also controls block moves to change map information. (See Chapter 4.)
9—10	PM	Previous Execution Level	This field defines the previous execution level. The code is interpreted as: **Code** — **Description** Bit 10 Bit 9 0 0 Kernel level 0 1 Executive level 1 0 Supervisor level 1 1 User level
11—12	CM	Current Execution Level	This field defines the current execution level. The code for bits 11 and 12 is interpreted in the same manner as that of bits 9 and 10 of the PM code, respectively. Changes to the CM field via instructions with the PSW as an explicit destination may affect the XMD pins during a prefetch access. Therefore, only microsequence instructions should be used to change the CM field state.

Table 2-1. Processor Status Word Fields (Continued)			
Bit(s)	**Field**	**Contents**	**Description**
13–16	IPL	Interrupt Priority Level	The IPL field represents the current interrupt priority level. Fifteen levels of interrupts are available. An interrupt, unless it is a nonmaskable interrupt, must have a higher priority level than the current IPL in order to be acknowledged. Therefore, level 0000 indicates that any of the fifteen interrupt priority levels (0001 through 1111) can interrupt the microprocessor. Level 1111, the highest interrupt priority level, indicates that no interrupts (except a nonmaskable interrupt) can interrupt the microprocessor.
17	TE	Trace Enable	This bit enables the trace function. When TE is set (1), it causes a trace trap to occur after execution of the next instruction. Debugging and analysis software use this facility for single-stepping a program. Changes to the TE field via instructions with the PSW as an explicit destination may cause unpredictable trace trap behavior (i.e., the instruction that changed the TE field in the PSW may or may not cause a trace trap). Therefore, only microsequence instructions should be used to change the TE field state.
18–21	NZVC	Condition Codes	The condition codes reflect the resulting status of the most recent instruction execution that affects them. These codes are tested using the conditional branch instructions and indicate the following when set (1): N - Negative (bit 21) V - Overflow (bit 19) Z - Zero (bit 20) C - Carry (bit 18)
22	OE	Enable Overflow Trap	This bit enables overflow traps when set (1). It is cleared (0) whenever an overflow trap is detected and handled.
23	CD	Cache Disable	This bit enables and disables the instruction cache. When the CD bit is cleared (0), the cache is used to store and read text. When the CD bit is set (1), the cache is not used. The instruction cache should only be disabled when its use could cause problems, e.g., when self-modifying code is executing. Changes to the CD field via instructions with the PSW as an explicit destination may corrupt the contents of the instruction cache. Therefore, only microsequence instructions should be used to change the CD field state.

Table 2-1. Processor Status Word Fields (Continued)			
Bit(s)	**Field**	**Contents**	**Description**
24	QIE	Quick-Interrupt Enable	The QIE enables and disables the quick-interrupt facility. If QIE is set (1), an interrupt is handled via the quick-interrupt sequence. If QIE is cleared (0), the interrupt causes a process switch (full-interrupt sequence).
25	CFD	Cache Flush Disable	When set (1), bit 25 disables instruction cache flushing (emptying of the cache's contents) when a new process is loaded via the XSWITCH_TWO microsequence (see **4.8.4 Microsequences**). When cleared (0), the contents of the cache are flushed when a new process is loaded via the XSWITCH_TWO microsequence.
26—31		Unused	These bits are not used and must always be cleared (0).

2.1.5 Stack Pointer

The stack pointer (SP), r12, contains the current 32-bit address of the top of the execution stack; i.e., the memory address of the next item to be stored on (pushed on) the stack or the last item retrieved (popped) from the stack. The stack pointer and the related instructions implement a LIFO (last-in-first-out) queue that supports efficient subroutine linkage and local variable storage.

2.1.6 Process Control Block Pointer

The process control block pointer (PCBP), r13, points to the starting address of the process control block for the current process. The process control block is a data structure in external memory that contains the hardware context of a process when the process is not running. This context consists of the initial and current contents of the processor status word, program counter, and stack pointer; the last contents of registers r0 through r10; boundaries for an execution stack; and block move specifications (and possibly memory specifications) for the process. The PCBP may only be written when the microprocessor is in kernel mode.

2.1.7 Interrupt Stack Pointer

The interrupt stack pointer (ISP), r14, contains the 32-bit memory address of the top of the interrupt stack. This stack is used when an interrupt request is received and also by the call process (CALLPS) and return to process (RETPS) instructions. The ISP may only be written when in kernel mode.

2.1.8 Program Counter

The program counter (PC), r15, contains the 32-bit memory address of the instruction being executed or, upon completion, the starting address of the next instruction to be executed. The PC may not be referenced in some addressing modes and is usually implicitly referenced by all program control instructions and all function calls and returns.

2.2 DATA HANDLING

All operations within the microprocessor are performed on 32-bit quantities, but data may be read or written as a byte, halfword, or word. Bits are numbered from right to left, starting at 0, and are right-adjusted on the address/data bus. The microprocessor automatically extends a byte or halfword to 32 bits before performing an operation. Zeros fill the high-order bits for unsigned operations, while the sign bit (bit 7 for bytes, bit 15 for halfwords) fills the high-order bits for signed operations. See Chapter 3 for a detailed description of data handling.

2.2.1 Data Types

The *WE* 32100 Microprocessor supports the following integer data types:

- byte — A byte is an 8-bit quantity that may appear at any address. Bits are numbered from right to left starting with 0, the least significant bit (LSB), and ending with 7, the most significant bit (MSB). (See Figure 2-4.)
- halfword — A halfword is a 16-bit quantity that may appear at any address that is divisible by 2. Bits are numbered from right to left starting with 0, the LSB, and ending with 15, the MSB.
- word — A word is a 32-bit quantity. A word may appear at any address that is divisible by 4. Bits are numbered right to left starting with 0, the LSB, and ending with 31, the MSB.

A bit field data type is also supported by the *WE* 32100 Microprocessor. A bit field is a sequence of 1 to 32 bits contained in a base word. The field is specified by the address of its base word, a bit offset, and a width. The *bit offset* ranges from 0 to 31 and identifies the starting bit of the field. The offset is numbered from the LSB of the base word and corresponds to the number of the bit in the word. That bit becomes bit 0, the LSB, of the field. The *width* ranges from 0 to 31 and specifies the size of the field. (Width plus one is the number of bits in the field.) The width is numbered from right to left in the field and corresponds to the bit number of the field's MSB. Fields do not extend across word boundaries and will wrap around from the MSB to LSB when the word boundary is reached. Figure 2-5 illustrates a bit field located at address **a**, with an offset of 6, and a width of 9. (Notice that the field contains 10 bits, one bit more than the width.)

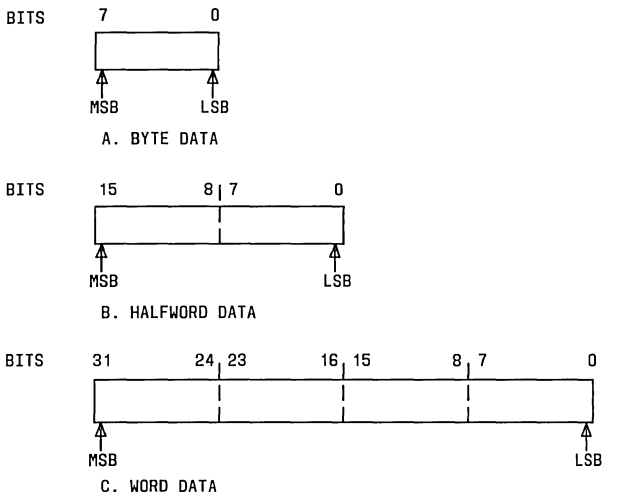


Figure 2-4. Bit Order of Data

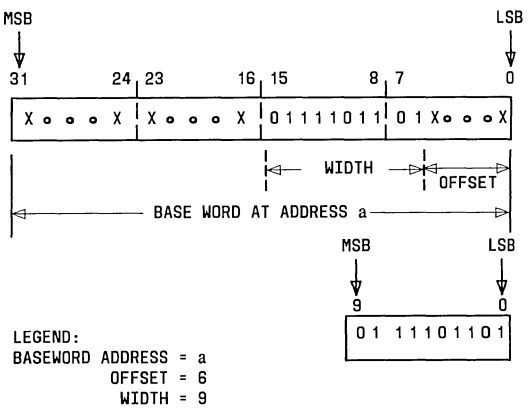


Figure 2-5. Bit Field Data Type

2.2.2 Data in Memory

Memory locations consist of a series of 8-bit (byte) locations for storing data. Halfwords occupy two consecutive memory locations and words occupy four consecutive memory locations. Boundary restrictions apply to the starting location of halfwords and words. Halfwords may only appear at addresses divisible by 2, and words may only appear at addresses divisible by 4. The microprocessor generates a fault if these boundaries are violated.

During memory reads the memory system must provide a word of data. The memory system must ignore the two lowest address bits (ADDR00 and ADDR01) and provide the word data beginning at this resulting word address.

Memory writes require that the memory system be set up in byte format, i.e., each byte must be writable independent of all other bytes. During memory writes, only the byte or bytes the CPU wants to write are to be changed. The remaining byte or bytes of the same word, if any, must not be altered. The CPU informs the memory system which byte(s) should be written based on the contents of the data size bits (DSIZE0 and DSIZE1) and the lower two address bits (ADDR00 and ADDR01). Table 2-2 indicates which byte(s) should be written based on the following byte addressing.

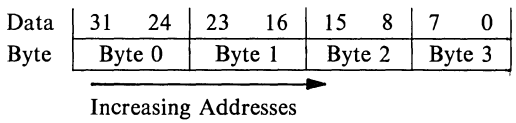


Table 2-2. Memory Write Summary

DSIZE1	DSIZE0		ADDR01	ADDR00	Memory Byte(s) Written: Byte 0	Byte 1	Byte 2	Byte 3
0	0	(Word)	0	0	Written	Written	Written	Written
1	0	(Halfword)	0	0	Written	Written	Unchanged	Unchanged
			1	0	Unchanged	Unchanged	Written	Written
1	1	(Byte)	0	0	Written	Unchanged	Unchanged	Unchanged
			0	1	Unchanged	Written	Unchanged	Unchanged
			1	0	Unchanged	Unchanged	Written	Unchanged
			1	1	Unchanged	Unchanged	Unchanged	Written

Note: For write transactions, any combination of DSIZE1—DSIZE0 and ADDR01—ADDR00 not indicated in the table generates an alignment fault.

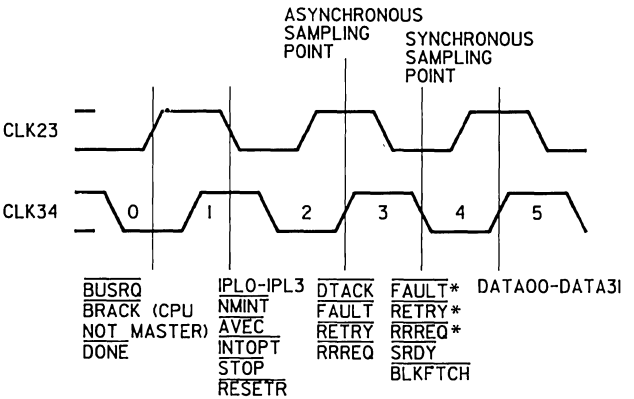
2.2.3 Memory Management

Memory management enables the operating system to efficiently manage the memory space for single and multiple processing applications. The memory management concepts are implemented with an external memory management unit (MMU) interfaced directly to the microprocessor. The MMU manipulates the microprocessor's vast address space by

accepting virtual addresses from the microprocessor and translating them into physical addresses (the physical address of the data). Therefore, the MMU can provide a vast address space per process (over four billion bytes of virtual or physical address space).

2.3 SIGNAL SAMPLING POINTS

The *WE* 32100 Microprocessor utilizes two phase-shifted input clocks (CLK23 and CLK34) as depicted on Figure 2-6. The CPU samples all inputs at the points indicated on this figure. This figure can be used as a reference for the protocol diagrams in the sections that follow.



* double latched

Notes:

1. BUSRQ, BRACK, IPL0—IPL3, NMINT, AVEC, INTOPT, STOP, RESETR are sampled repetitively one CLK34 cycle apart (i.e., on every clock cycle).
2. After DTACK is asserted, FAULT, RETRY, RRREQ and BLKFTCH are sampled once at the synchronous sampling point. If FAULT, RETRY, or RRREQ are asserted prior to or at the same time as DTACK, then they are sampled once and double latched. If SRDY is asserted, then FAULT, RETRY, RRREQ and BLKFTCH are sampled once at the synchronous sampling point.
3. BLKFTCH must remain asserted until negation of data strobe (DS).
4. DSHAD is not latched and can be asserted at any time subject to the following conditions: DSHAD should only be asserted during a CPU-initiated transaction while AS is active and DTACK, SRDY, and FAULT are inactive. Unless RETRY or RRREQ is active, DSHAD should only be negated while AS is still active and DTACK, SRDY and FAULT are inactive. If RETRY or RRREQ is active, then DSHAD should be negated one cycle after AS is negated.

Figure 2-6. Signal Sampling Points

The bus transactions that are described in the upcoming sections share the following attributes. The read/write (R/$\overline{W}$) output remains in its mode (high, logic 1, for read transactions and low, logic 0, for write transactions) for the entire transaction. The cycle initiate ($\overline{CYCLEI}$) output goes active for two clock cycles at the beginning of each transaction. The CPU asserts the data ready ($\overline{DRDY}$) output at the end of the transaction if there are no bus exceptions (fault, $\overline{FAULT}$; retry, $\overline{RETRY}$; or relinquish and retry ($\overline{RRREQ}$) during the transaction.

The address bus (ADDR00—ADDR31) is driven for the entire transaction if the CPU is operating in physical mode. If the CPU is operating in virtual mode, the CPU only drives the address bus during the first and second clock states. (One clock state is half a clock cycle.) The CPU 3-states its address bus during the third clock state so that the MMU can drive the translated physical address onto the bus.

The data size bits (DSIZE0—DSIZE1) indicate the size of the transaction (byte, halfword, word, or double word) and are driven for the entire transaction. The access status code (SAS0—SAS3) is driven one clock cycle before the transaction starts and remains active for two additional cycles during the transaction. This 4-bit code indicates the type of transaction being performed. At clock state four, the access status code is changed to reflect the next operation (if the next operation is a bus transaction) or to "no operation" (if the next operation is not a bus transaction). The leading edge of $\overline{CYCLEI}$ can be used to latch the access status code.

2.4 READ AND WRITE OPERATIONS

The *WE* 32100 Microprocessor performs zero wait-state read and write accesses in three clock cycles. These accesses are performed in two stages. The microprocessor first outputs the address and the control signals necessary for the given operation. Once these signals have had time to settle, the data transfer takes place. All accesses are followed by a vestigial cycle to allow enough time for a memory management unit to release the shared address bus.

Two inputs that allow handshaking between the CPU and slow slave devices are provided. External devices can cause the CPU to insert wait cycles during a bus transaction through the use of the synchronous ready ($\overline{SRDY}$) input and the data transfer acknowledge ($\overline{DTACK}$) input. Wait cycles prolong a bus transaction which allows slave devices more time to place data on the bus during a read transaction and more time to pick up data from the bus during a write transaction.

During bus transactions the CPU samples the $\overline{DTACK}$ and $\overline{SRDY}$ inputs at their respective sampling points, as shown on Figure 2-6. If either input is active (low) at its sampling point, no wait cycles are inserted and the transaction completes in three clock cycles. However, if neither $\overline{DTACK}$ nor $\overline{SRDY}$ is sampled active, wait cycles are inserted, and the CPU samples each input at cycle intervals from its respective sampling point until either input is sampled active. At this point no more new wait cycles are generated and the bus transaction completes one clock cycle after the completion of the current wait cycle.

In the following read and write operation descriptions, the term "asserted" means that a signal is driven to its active state either by the microprocessor (outputs) or by an external device (inputs). The term "negated" means that the signal is driven to its inactive state. A bar over a signal name (e.g., $\overline{\text{AS}}$) indicates that the signal is active low, logic 0.

2.4.1 Read Transaction Using $\overline{\text{SRDY}}$

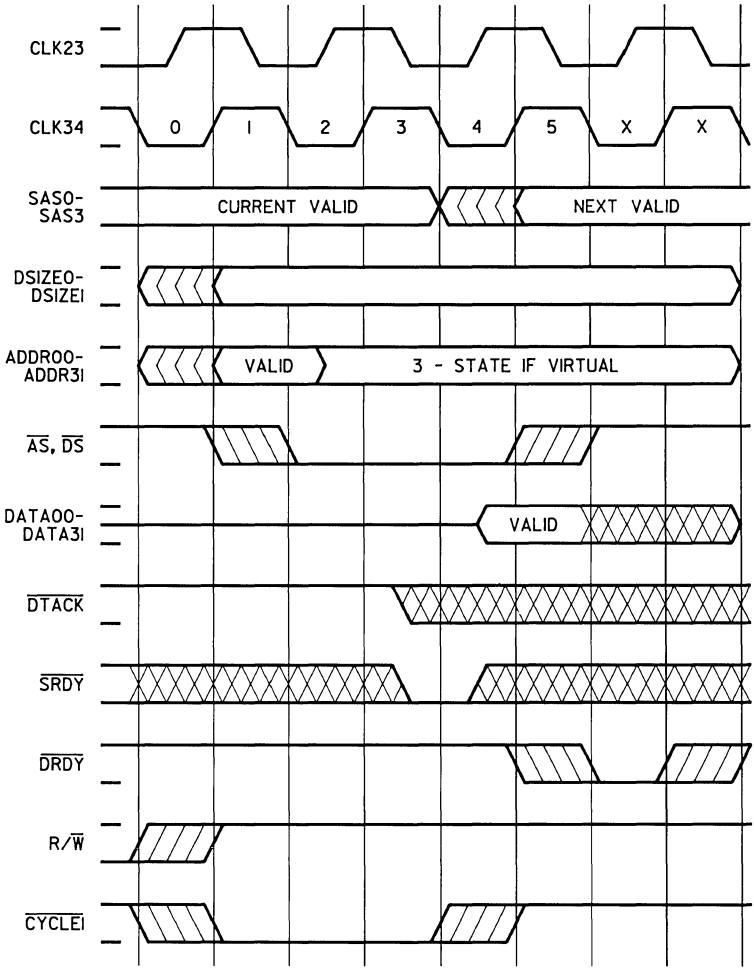
Figure 2-7 illustrates a read transaction with zero wait cycles (3 cycle access) using $\overline{\text{SRDY}}$ to terminate the access. The read transaction starts with the CPU driving the address bus (ADDR00—ADDR31) and the data size outputs (DSIZE0—DSIZE1), negating the read/write (R/$\overline{\text{W}}$) output to indicate that a read operation is being performed, and asserting the cycle initiate ($\overline{\text{CYCLE1}}$) output at the beginning of clock state zero.

For read operations the address strobe ($\overline{\text{AS}}$) and data strobe ($\overline{\text{DS}}$) have the same timing. The CPU latches data driven onto the data bus by the addressed device at the end of clock state four, when the CPU negates $\overline{\text{AS}}$ and $\overline{\text{DS}}$. Data can be driven onto the bus while $\overline{\text{AS}}$ and $\overline{\text{DS}}$ are active.

The transaction illustrated on Figure 2-7 is terminated by the assertion of $\overline{\text{SRDY}}$ by the addressed device. $\overline{\text{SRDY}}$ is the acknowledgement that the addressed device is putting the data onto the data bus and that the CPU can latch the data and terminate the transaction.

$\overline{\text{SRDY}}$ is synchronously sampled at the end of clock state three.

The read transaction depicted on Figure 2-7 completes in three clock cycles (zero wait cycles) because $\overline{\text{SRDY}}$ is active when sampled at the end of the clock state three.



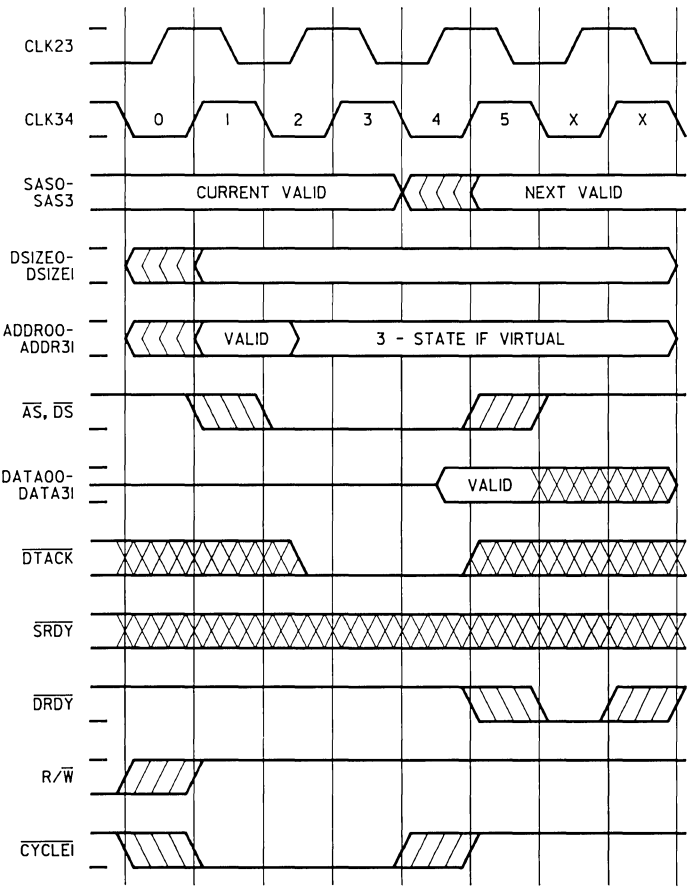
Note: Zero Wait Cycles.

Figure 2-7. Read Transaction (Using SRDY)

2.4.2 Read Transaction Using DTACK

The read transaction using DTACK is identical to the read transaction using SRDY, except that the addressed device asserts DTACK to acknowledge that it is putting data on the data bus instead of SRDY (see Figure 2-8). DTACK is asynchronously sampled at the end of clock state two and is double latched to avoid metastability.

The read transaction shown on Figure 2-8 completes in three clock cycles because the CPU samples DTACK active at the end of clock state two. Upon sampling DTACK active, the CPU latches the data and terminates the transaction.



Note: Zero Wait Cycles.

Figure 2-8. Read Transaction (Using DTACK)

2.4.3 Read Transaction With Wait Cycle Using SRDY

The CPU inserts wait cycles during bus transactions if it does not sample DTACK active at the end of clock state two or SRDY active at the end of clock state three, and no bus exceptions occur. As illustrated on Figure 2-9, the CPU inserts one wait cycle because DTACK is not active at the end of clock state two and SRDY is not active at the end of clock state three. Only one wait cycle is inserted during the transaction because SRDY is active when sampled at the end of the wait cycle. The CPU then latches the data and terminates the transaction.

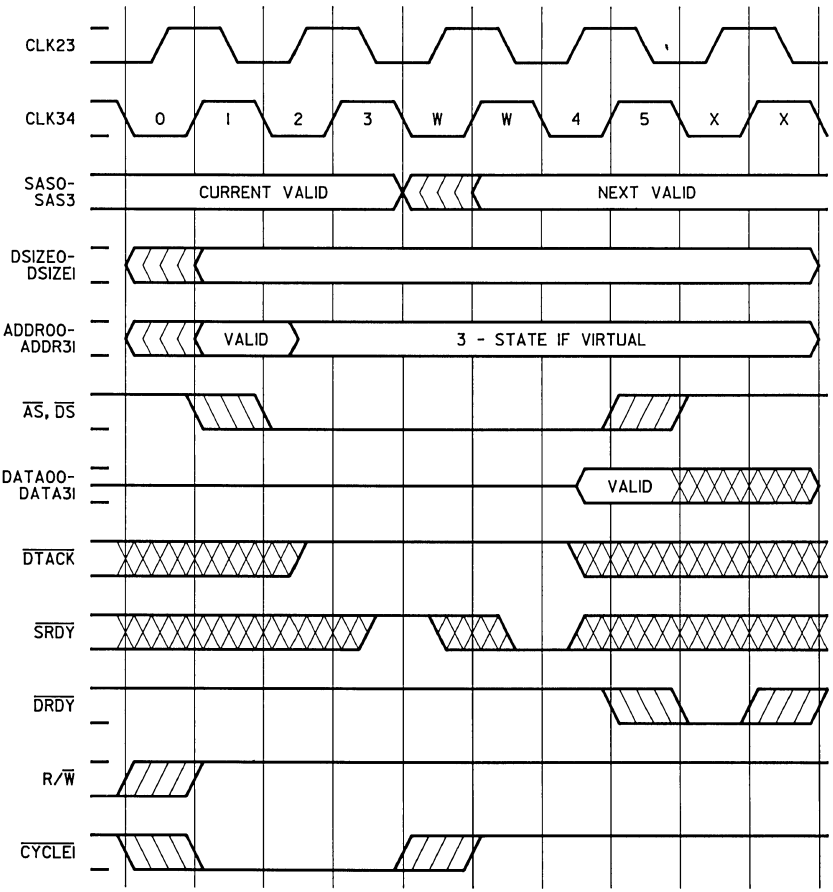


Figure 2-9. Read Transaction With One Wait Cycle (Using SRDY)

2.4.4 Read Transaction With Two Wait Cycles Using DTACK

The CPU can insert multiple wait cycles during bus transactions, as illustrated on Figure 2-10. In this figure the CPU does not receive an acknowledge (DTACK or SRDY) for two clock cycles. Neither DTACK nor SRDY is active during clock states two and three or the first wait cycle. DTACK is sampled active in the middle of the second wait cycle, causing the termination of wait cycle generation. The CPU then latches the data and terminates the transaction.

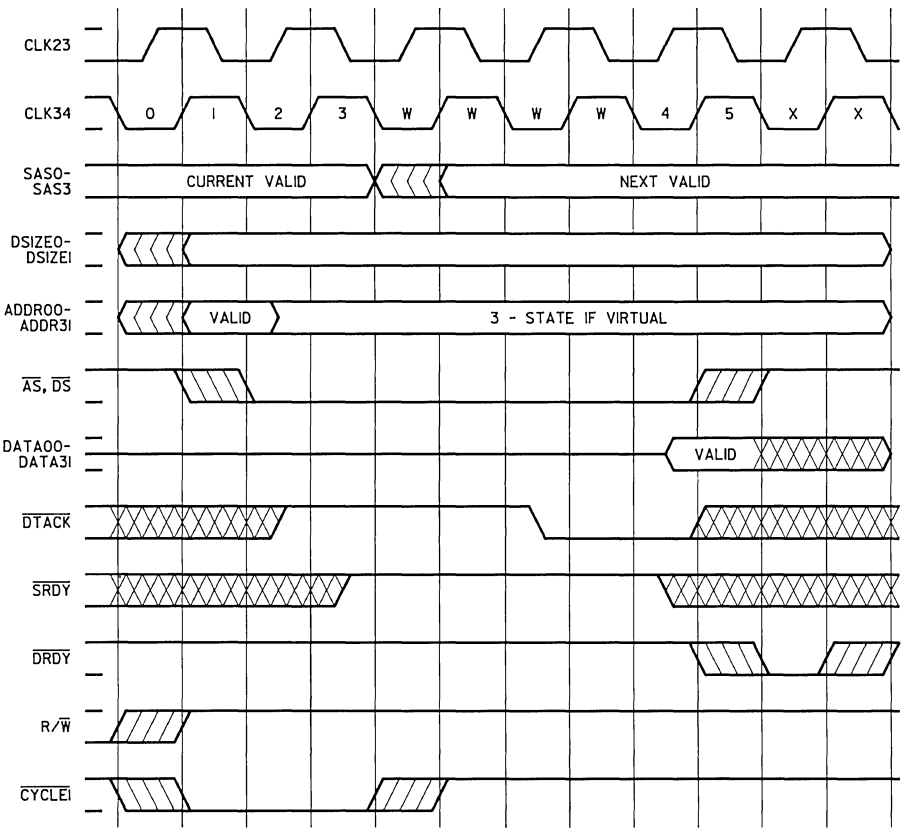


Figure 2-10. Read Transaction With Two Wait Cycles (Using DTACK)

2.4.5 Write Transaction Using SRDY

During write transactions the R/W output is held low (logic 0) for the entire transaction. The CPU drives the data bus with the data to be written from clock state two until the end of the transaction. The access status code at the beginning of a write transaction is "write" (SAS3–SAS0 = 1010).

Unlike read transactions where AS and DS have the same timing, the CPU asserts DS one cycle after it has asserted AS, allowing the addressed device to latch the data with either the leading or trailing edge of DS.

Figure 2-11 illustrates a write transaction with the addressed device using SRDY as the acknowledgement. By asserting SRDY the addressed device indicates to the CPU that it is ready to latch the data on the data bus. SRDY is synchronously sampled at the end of clock state three. On Figure 2-11, the CPU sampled DTACK inactive at the end of clock state two; however, it sampled SRDY active at the end of clock state three. As a result, the CPU terminates the transaction.

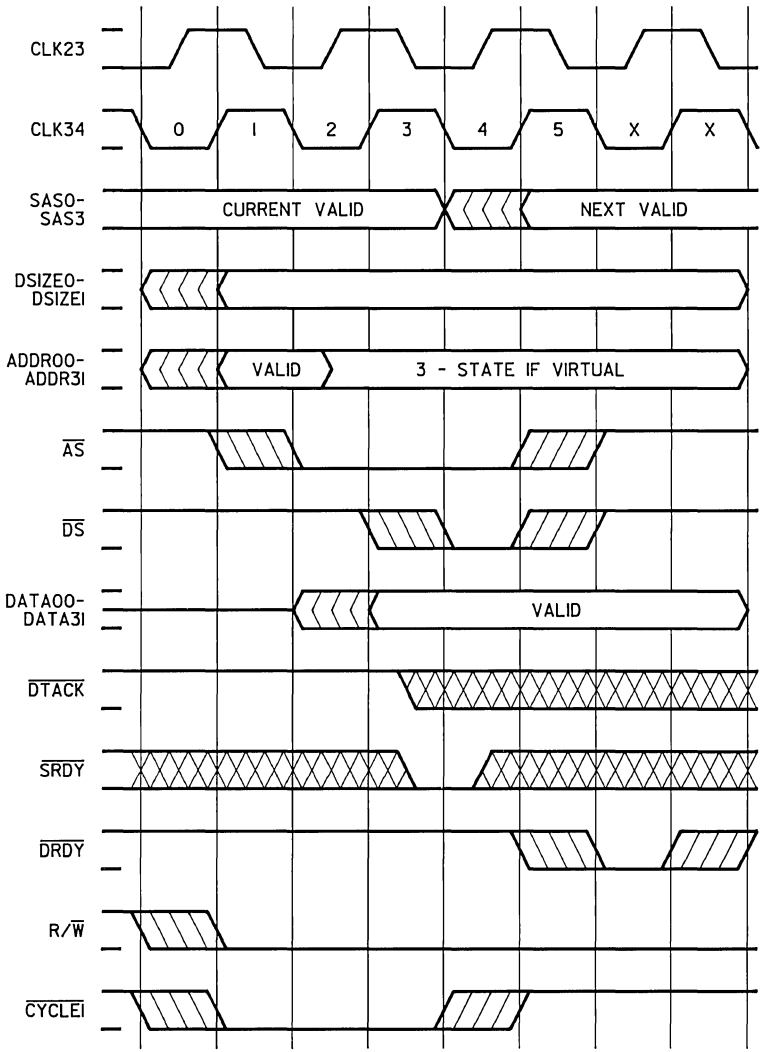
2.4.6 Write Transaction Using DTACK

The write transaction using DTACK is identical to the write transaction using SRDY, except that the addressed device asserts DTACK to indicate that it is ready to latch the data on the data bus. DTACK is sampled asynchronously at the end of clock state two. On Figure 2-12,the CPU samples DTACK active at this time and proceeds to terminate the transaction.

2.4.7 Write Transaction With Wait Cycle Using SRDY

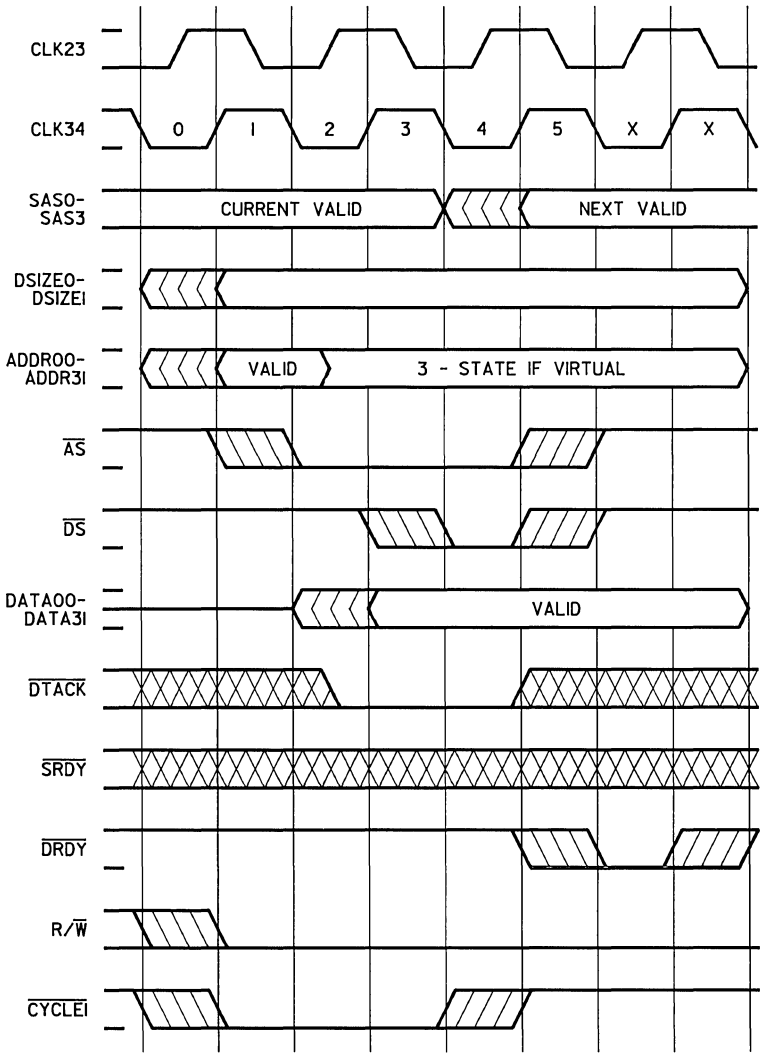
Wait cycle insertion for write transactions is similar to wait cycle insertion for read transactions. Just as in read transactions, the CPU inserts wait cycles if DTACK is not active when sampled at the end of clock state two, SRDY is not active when sampled at the end of clock state three, and no bus exceptions occur.

Figure 2-13 illustrates a write transaction with two wait cycles. The CPU begins wait cycle insertion because DTACK is not active at the end of state two and SRDY is not active at the end of state three. A second wait cycle is inserted because, again, neither input was active when sampled during the first wait cycle. The addressed device finally asserts SRDY at the end of the second wait cycle, and the CPU terminates the transaction.



Note: Zero wait cycles.

Figure 2-11. Write Transaction (Using SRDY)



Note: Zero wait cycles.

Figure 2-12. Write Transaction (Using DTACK)

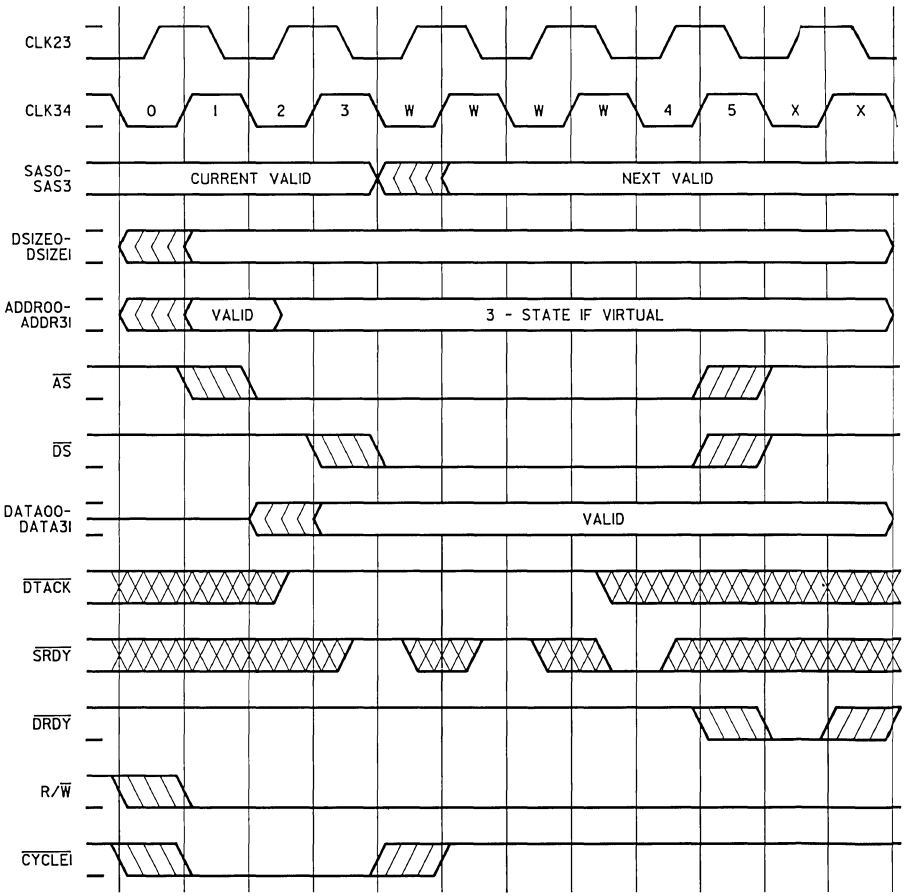


Figure 2-13. Write Transaction With Two Wait Cycles (Using $\overline{\text{SRDY}}$)

2.4.8 Write Transaction With Wait Cycle Using $\overline{\text{DTACK}}$

The write transaction shown on Figure 2-14 is another example of wait cycle insertion. In this transaction the addressed device asserts $\overline{\text{DTACK}}$ to indicate that it is ready to latch the data, and that therefore, no more wait cycles are to be inserted.

Neither $\overline{\text{DTACK}}$ nor $\overline{\text{SRDY}}$ is active at its initial sampling point and, as a result, the CPU inserts a wait cycle. When the CPU samples $\overline{\text{DTACK}}$ a second time during the wait cycle, $\overline{\text{DTACK}}$ is now active. The CPU can then terminate the transaction.

Additional protocol diagrams for read and write operations are included in **2.19 Supplementary Protocol Diagrams**.

2.5 READ INTERLOCKED OPERATION

Read interlocked operation consists of a memory fetch (read access) and one or more internal microprocessor operations, followed by a write access to the same memory location. Once the read access has been completed, the read interlocked operation may not be preempted other than by a reset. This prevents another process from altering data in memory which is being operated on by the current process. If a fault occurs during the read access, the read interlocked operation terminates without going through the write access.

Figure 2-15 illustrates a read interlocked transaction. Note that the access status code is "read interlocked" (SAS3—SAS0 = 0111) for both transactions and that the address remains the same for both transactions. The read portion and the write portion of the transaction are standard read and write transactions.

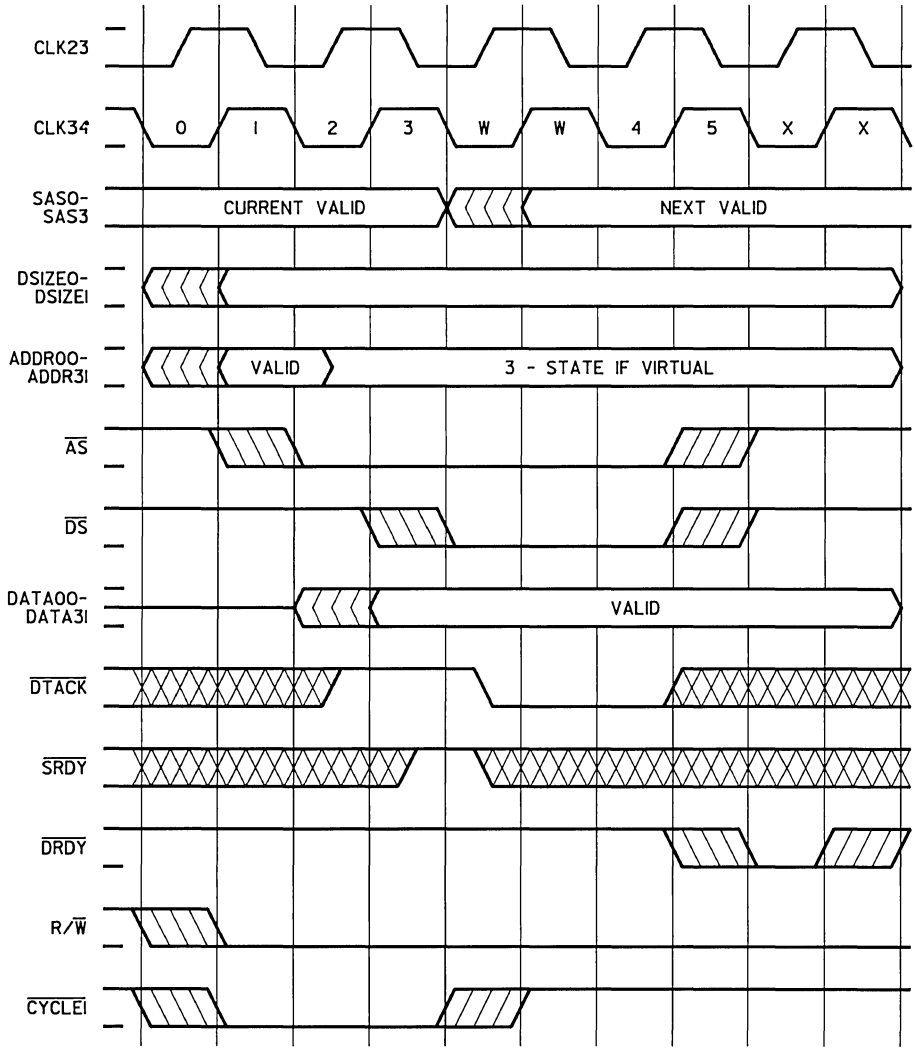
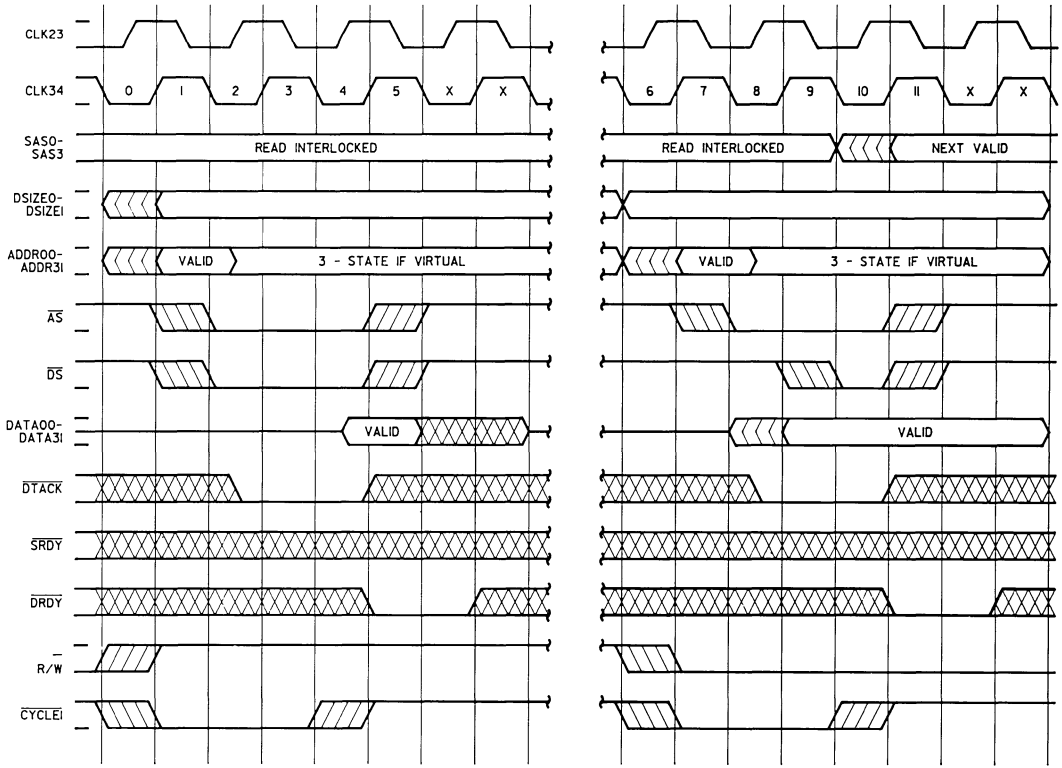


Figure 2-14. Write Transaction With One Wait Cycle (Using $\overline{\text{DTACK}}$)



Notes:

1. Number of cycles between the read transaction and write transaction is four for swap word interlocked (SWAPWI) and six for swap halfword interlocked (SWAPHI) and swap byte interlocked (SWAPBI) instructions.
2. Zero wait cycles.

Figure 2-15. Read Interlocked Transaction (Using $\overline{\text{DTACK}}$)

2.6 BLOCKFETCH OPERATION

The CPU can fetch two words of instruction code in one bus transaction via a blockfetch operation. The CPU generates one address, and the memory provides two words of instruction code. This reduces the number of cycles that it takes to fetch two words. The CPU starts the transaction with the DSIZE of double word, which indicates that it is ready to perform a blockfetch.

If the memory is designed to handle blockfetch, it will respond with the blockfetch (BLKFTCH) signal and an acknowledge signal, either SRDY or DTACK.

2.6.1 Blockfetch Transaction Using SRDY

After the memory issues BLKFTCH and SRDY, the CPU latches the data being sourced by the memory during clock state four, removes DS, and keeps AS in the active state. One cycle later the CPU reissues DS and is ready to latch the second word.

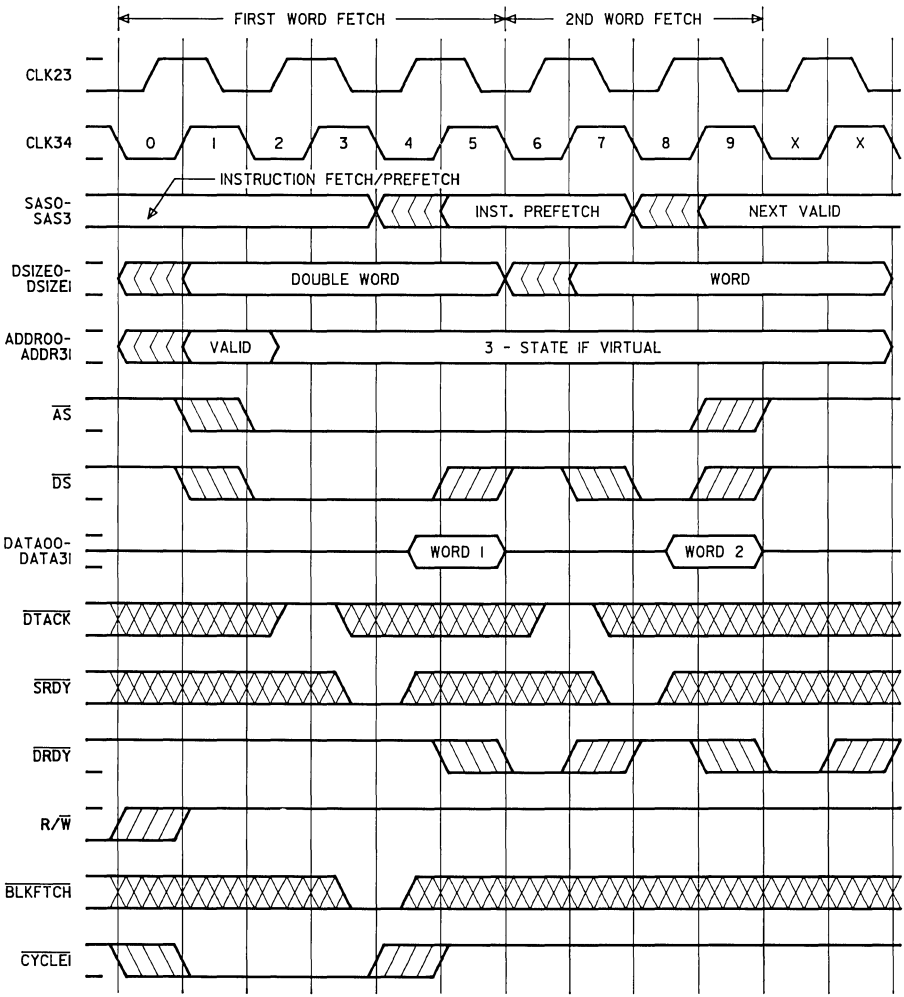
The memory drives the data bus with the second word and a SRDY. The CPU samples the SRDY at the end of clock state seven, then latches the data during clock state eight and terminates the transaction. This operation is shown on Figure 2-16.

AS stays low for both words fetched. DS goes inactive for one cycle in between the first and second words. DSIZE changes from double word to word at clock state six. R/W is held in the read mode for the entire transaction. Only one CYCLE1 is issued for this transaction. Two DRDY's are issued, one for each word. The BLKFTCH pin is sampled only with the first SRDY. It is not used during the second word. The SAS code for the first word can be "instruction fetch," "instruction fetch after PC discontinuity," or "prefetch." SAS for the second word is always "prefetch." If the memory does not issue a BLKFTCH with the acknowledgement on the first word then the CPU will latch the data and terminate the transaction by removing both AS and DS. It will then precede to start up a second read with SAS of "prefetch" and issue a new address.

For a blockfetch transaction, the CPU issues only one address. If it is in virtual mode the CPU 3-states the address during clock state two which allows the MMU to drive the physical address for fetching both words. Also note that the CPU is fetching the two words from a double word address block. It will ask for either the even or odd address first, as indicated by the value on the address bus. For the second word it expects the memory to provide the data corresponding to the address location with ADDR02 complemented. This is the other corresponding word from the double word block.

For example, assuming physical addressing, the CPU drives ADDR with 0003C000. Memory provides data for the first word corresponding to location 0003C000. Memory provides data for the second word corresponding to location 0003C004.

As another example of physical mode, consider the CPU driving ADDR with 00078004. Memory provides data for the first word corresponding to location 00078004. Memory provides data for the second word corresponding to location 0007800.

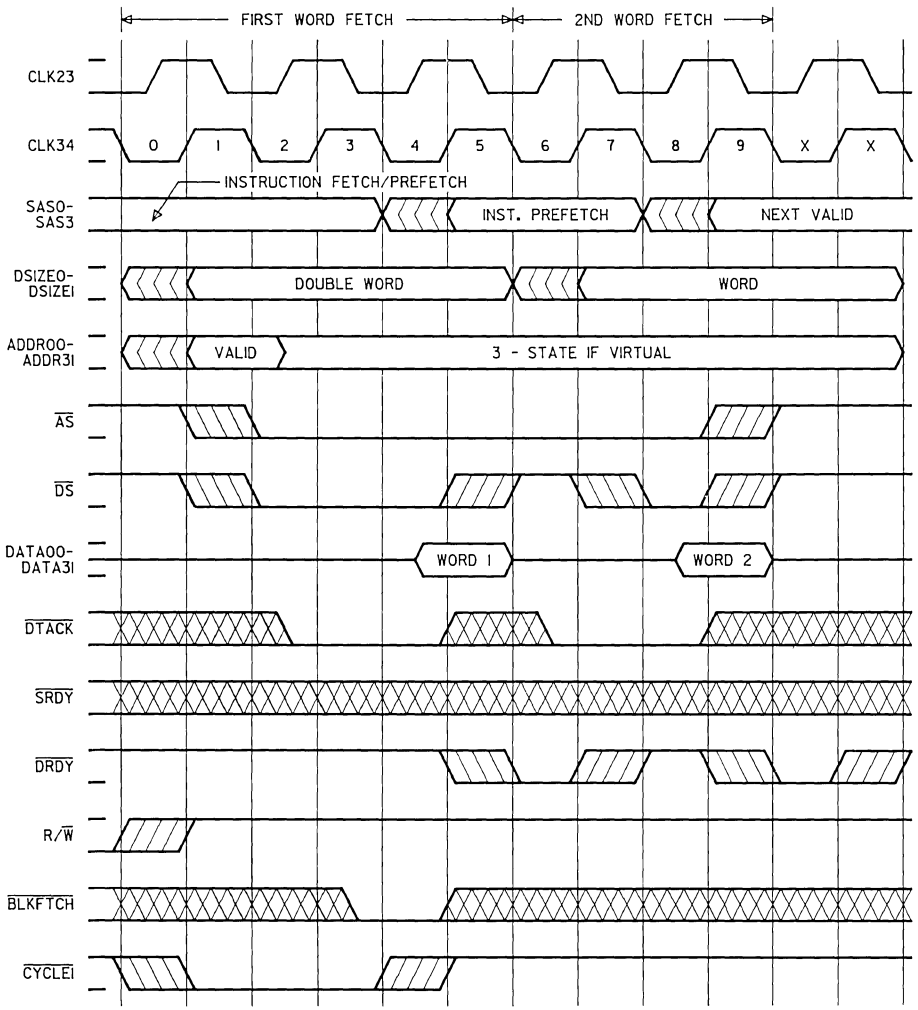


Note: Zero wait cycles.

Figure 2-16. Blockfetch Transaction (Using SRDY)

2.6.2 Blockfetch Transaction Using $\overline{\text{DTACK}}$

This transaction (see Figure 2-17) is the same as Figure 2-16 except the acknowledgement used by the memory is $\overline{\text{DTACK}}$. On the first word $\overline{\text{DTACK}}$ is sampled by the CPU at the end of clock state two. $\overline{\text{BLKFTCH}}$ is sampled at the end of clock state three which is the same as on Figure 2-16. For the second word, $\overline{\text{DTACK}}$ is sampled at the end of clock state six.

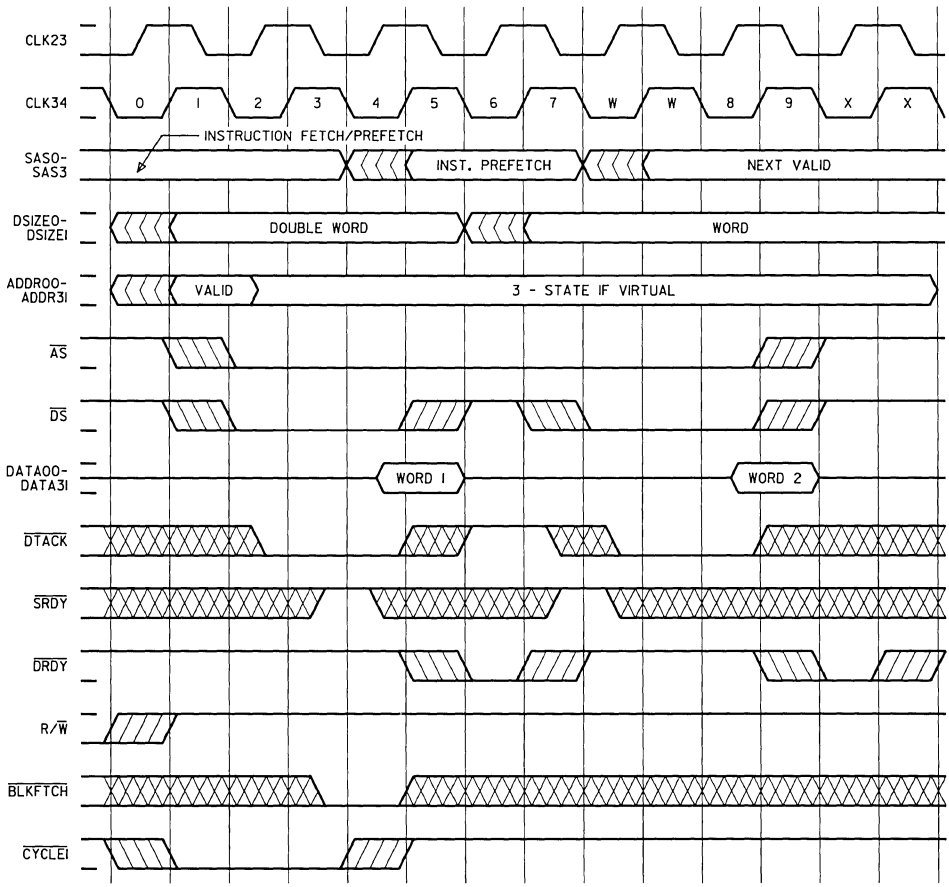


Note: Zero wait cycles.

Figure 2-17. Blockfetch Transaction (Using $\overline{\text{DTACK}}$)

2.6.3 Blockfetch Transaction Using DTACK With Wait Cycle on Second Word

In this case (see Figure 2-18), during the fetch of the second word there was no DTACK during clock state six nor SRDY during clock state seven. Therefore, the CPU inserted a wait cycle. It sampled DTACK during the first clock state "W," then latched the data and terminated the transaction.

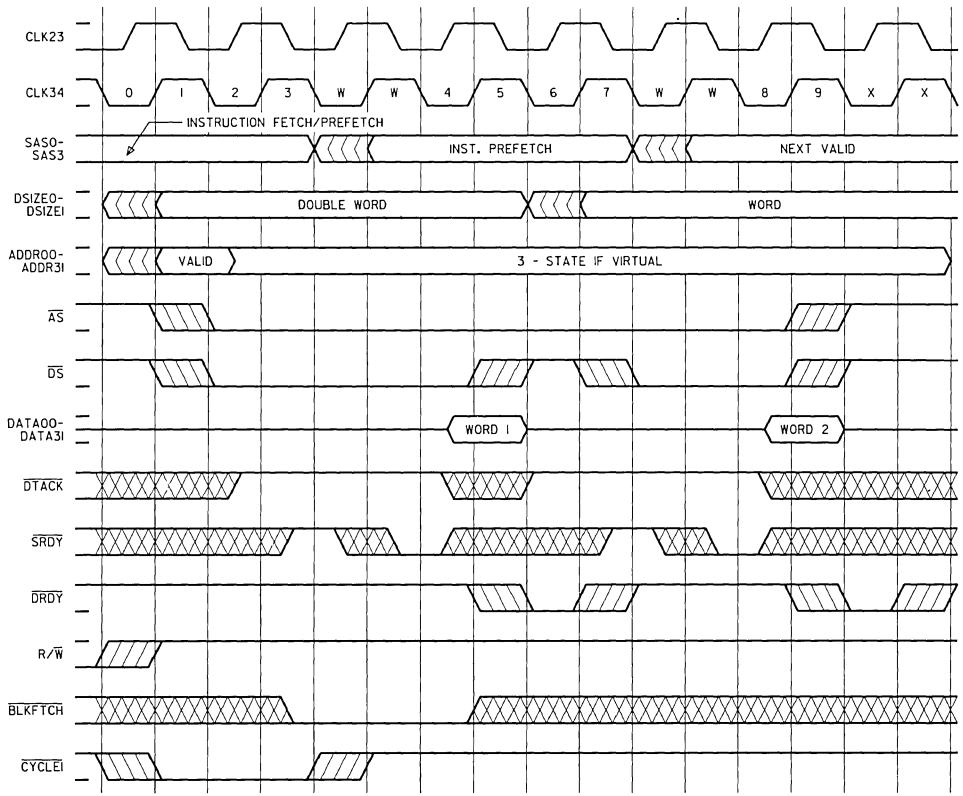


Note: Wait cycle on second word.

Figure 2-18. Blockfetch Transaction Using DTACK With Wait Cycle on Second Word

2.6.4 Blockfetch Transaction Using SRDY With Wait Cycles on Both Words

During the fetch of the first word in this transaction, the CPU did not sample a DTACK during clock state two nor a SRDY during clock state three (see Figure 2-19). It inserted a wait cycle. During the second clock state "W," it sampled SRDY and BLKFTCH, then latched the data and terminated DS. The CPU proceeded to the second fetch. During clock state six it did not sample DTACK nor a SRDY during clock state seven. The CPU inserted a wait cycle. During the second clock state "W," it sampled SRDY, then latched the data and terminated the transaction.



Note: Wait cycle on both words.

Figure 2-19. Blockfetch Transaction (Using SRDY)

2.7 BUS EXCEPTIONS

Bus exceptions cause the termination of the current memory access and result when an access retry is required or when a fault occurs during an access. The three bus exceptions are fault, retry, and relinquish and retry.

A fault is the result of an error condition during a bus cycle. An external device reports errors to the CPU (such as address translation and memory faults) by asserting the fault input ($\overline{\text{FAULT}}$). This causes the CPU to terminate the access and perhaps execute a fault handling routine. The *WE* 32101 Memory Management Unit uses the $\overline{\text{FAULT}}$ input when it detects that a virtual address corresponds to data that is not presently in physical memory. The MMU also generates a fault if it detects an error condition when it attempts to translate the virtual address. A retry causes the CPU to retry the access. An external device requests a retry by asserting the $\overline{\text{RETRY}}$ input. A relinquish and retry causes the microprocessor to give up its bus and retry the preempted access once the bus has been returned to its control. An external device requests a relinquish and retry by asserting the $\overline{\text{RRREQ}}$ input.

Table 2-3 describes how the microprocessor handles the simultaneous assertion of two or more bus exceptions. The term *negated* indicates the signal is driven to its inactive state.

2.7.1 Faults

A bus transaction can be terminated by a bus exception: in this case, $\overline{\text{FAULT}}$ without a $\overline{\text{DTACK}}$ or $\overline{\text{SRDY}}$ (see Figure 2-20). On Figure 2-20, the CPU inserted two wait cycles

Table 2-3. Simultaneously Asserted Exception Conditions*

Simultaneously Asserted Signals	Behavior
$\overline{\text{RRREQ}}$, $\overline{\text{RETRY}}$, $\overline{\text{FAULT}}$	The relinquish and retry request ($\overline{\text{RRREQ}}$) is honored first. The microprocessor acknowledges this request by relinquishing the bus and then asserting the relinquish and retry request acknowledge ($\overline{\text{RRRACK}}$) output. The access is retried once $\overline{\text{RRREQ}}$ and $\overline{\text{RETRY}}$ are negated by the requesting devices. If the fault ($\overline{\text{FAULT}}$) input is still asserted during the retried access, the fault will be honored (recognized). The fault input will be recognized only during the retried access.
$\overline{\text{RRREQ}}$, $\overline{\text{RETRY}}$	The relinquish and retry request ($\overline{\text{RRREQ}}$) is honored first. The microprocessor 3-states the appropriate signals and then asserts the relinquish and retry acknowledge output ($\overline{\text{RRRACK}}$). The access is retried once $\overline{\text{RRREQ}}$ and $\overline{\text{RETRY}}$ are negated.
$\overline{\text{RRREQ}}$, $\overline{\text{FAULT}}$	Same as in behavior for $\overline{\text{RRREQ}}$, $\overline{\text{RETRY}}$, and $\overline{\text{FAULT}}$ simultaneously asserted.
$\overline{\text{RETRY}}$, $\overline{\text{FAULT}}$	The $\overline{\text{RETRY}}$ request is honored first. The $\overline{\text{FAULT}}$ will be recognized on the retried access if it is still asserted.

* Table 2-3 applies only when the microprocessor is the bus master.

because it did not receive an acknowledge or a bus exception. During the third clock state "W," the CPU asynchronously sampled FAULT and terminated the transaction. Note that if a DTACK was also sampled with the FAULT during the third clock state "W," the figure would still look identical.

Upon the faulted transaction, the CPU will proceed to the fault handler to process the exception. However, for a faulted prefetch, the CPU ignores the data, continues with its current instruction execution, and does not enter the fault handler. If the CPU needs this instruction later, it will do an instruction fetch, and if this is also faulted, the CPU will proceed with the fault handler. At the end of the transaction, DRDY is not issued starting with clock state five because the CPU sampled the FAULT. If this bus transaction is a write, the CPU will sample FAULT the same way as in a read case. There are some differences for a blockfetch which can be seen in **2.8 BLOCKFETCH SPECIAL CASES.**

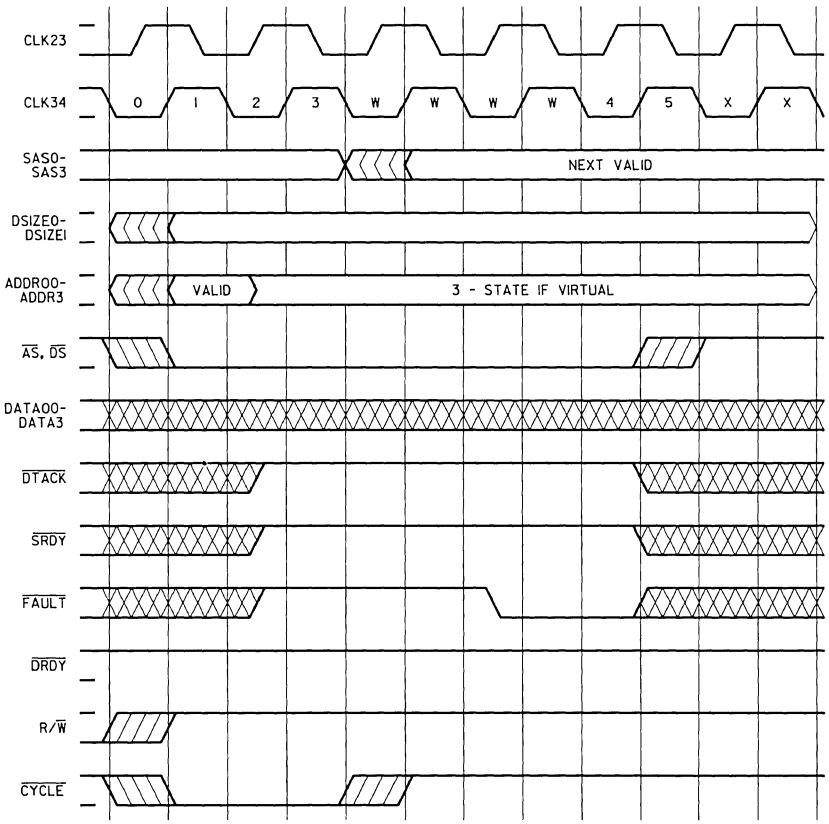


Figure 2-20. Asynchronous Fault Without DTACK and SRDY (Read Transaction)

FAULT With SRDY

The CPU can sample FAULT synchronously if it has a SRDY with it. Figure 2-21 shows both SRDY and FAULT being sampled during the last clock state "W." The CPU then terminates the transaction and does not issue a DRDY. For reads and writes, the CPU samples the SRDY and FAULT in the same way.

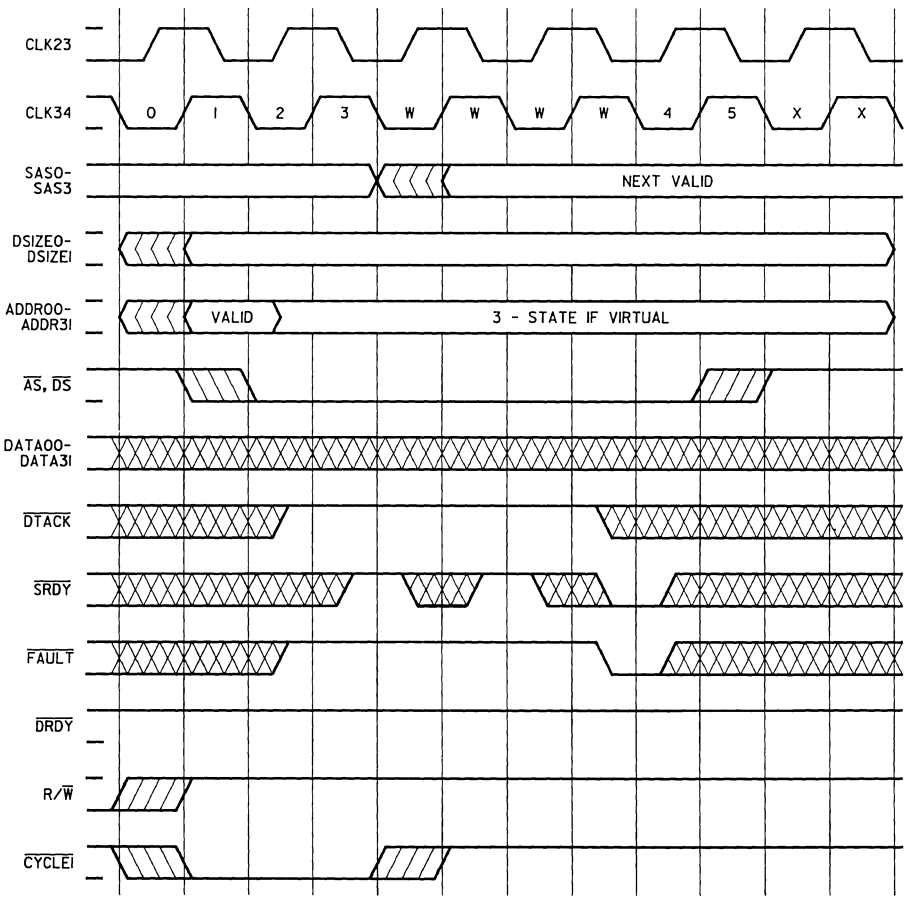
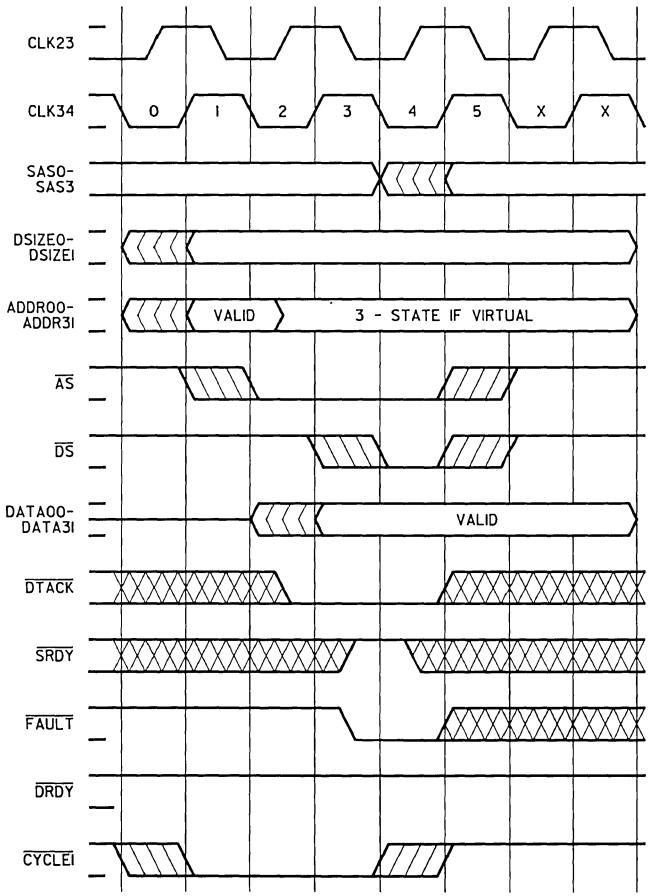


Figure 2-21. Fault with Synchronous Ready (SRDY); i.e., Synchronous Fault

FAULT After DTACK

The CPU can also sample FAULT synchronously if it has sampled a DTACK asynchronously in the same clock cycle. Figure 2-22 shows DTACK sampled during clock state three and FAULT sampled during clock state four. The CPU then terminates the bus and does not issue DRDY. This sampling of STACK and FAULT is the same for reads.



Note: FAULT must meet setup time with respect to CLK34 edge after assertion of DTACK.

Figure 2-22. Fault After Assertion of DTACK (Write Transaction is Shown)

2.7.2 Retry

$\overline{\text{RETRY}}$ is sampled the same way the $\overline{\text{FAULT}}$ is sampled. The previous figures on how $\overline{\text{FAULT}}$ is sampled can have the words $\overline{\text{FAULT}}$ replaced by $\overline{\text{RETRY}}$ as far as the sampling is concerned. Figure 2-23 shows a retry for a read transaction.

When the CPU samples the $\overline{\text{RETRY}}$, it terminates the transaction and does not issue a $\overline{\text{DRDY}}$. The CPU continues to asynchronously sample $\overline{\text{RETRY}}$. After $\overline{\text{RETRY}}$ is removed, the CPU will redo the entire transaction. The SAS code will be the same as the first transaction as well as the address, DSIZE, and R/$\overline{\text{W}}$.

$\overline{\text{RETRY}}$ operates on reads and writes in the same way. There are some differences if the transaction is a blockfetch, and these can be seen in the $\overline{\text{RETRY}}$ with blockfetch figures in **2.8 BLOCKFETCH SPECIAL CASES**.

2.7.3 Relinquish and Retry

$\overline{\text{RRREQ}}$ is sampled the same way that the other two bus exceptions ($\overline{\text{FAULT}}$ and $\overline{\text{RETRY}}$) are sampled. An example is shown on Figure 2-24.

When the CPU samples $\overline{\text{RRREQ}}$, it terminates the transaction and does not issue a $\overline{\text{DRDY}}$. After the second clock state "X," the CPU 3-states the address and data buses as well as most of the control signals in order to allow some other device to use the bus. A cycle later the CPU issues $\overline{\text{RRRACK}}$. This indicates that the device that had issued $\overline{\text{RRREQ}}$ can get onto the bus and do its bus transaction. The CPU will continue to asynchronously sample $\overline{\text{RRREQ}}$. When the device using the bus is finished, it should remove $\overline{\text{RRREQ}}$. When the CPU sees that $\overline{\text{RRREQ}}$ is removed, it will take back the bus and redo the entire transaction. As in the $\overline{\text{RETRY}}$ case, the SAS code, address, DSIZE, and R/$\overline{\text{W}}$ will be the same on both transactions.

$\overline{\text{RRREQ}}$ operates on reads and writes in the same way. There are some differences if the transaction is a blockfetch, and these can be seen in the blockfetch with relinquish and retry figure in **2.8. BLOCKFETCH SPECIAL CASES**.

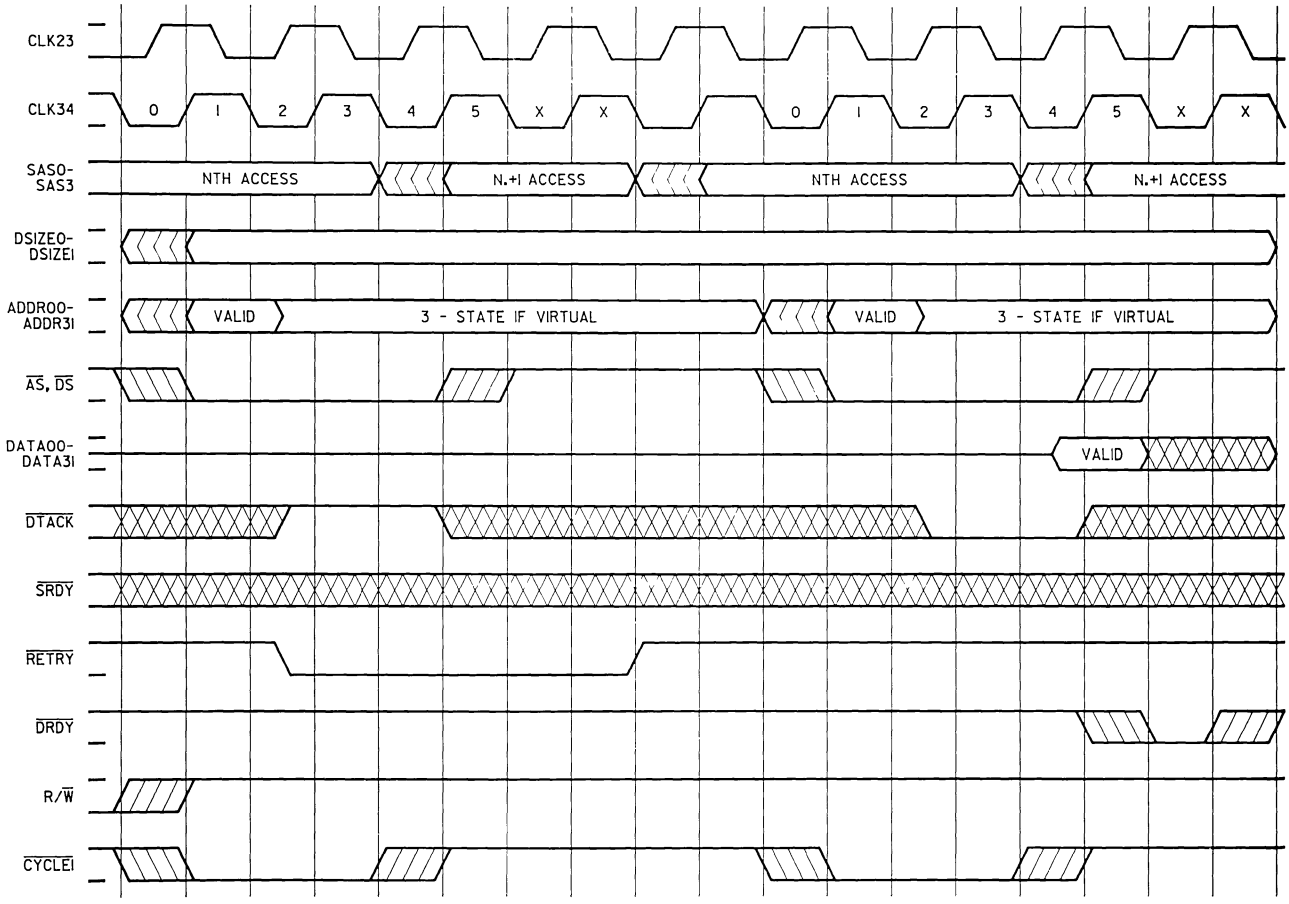


Figure 2-23. Retry of Transaction (Read Transaction is Shown)

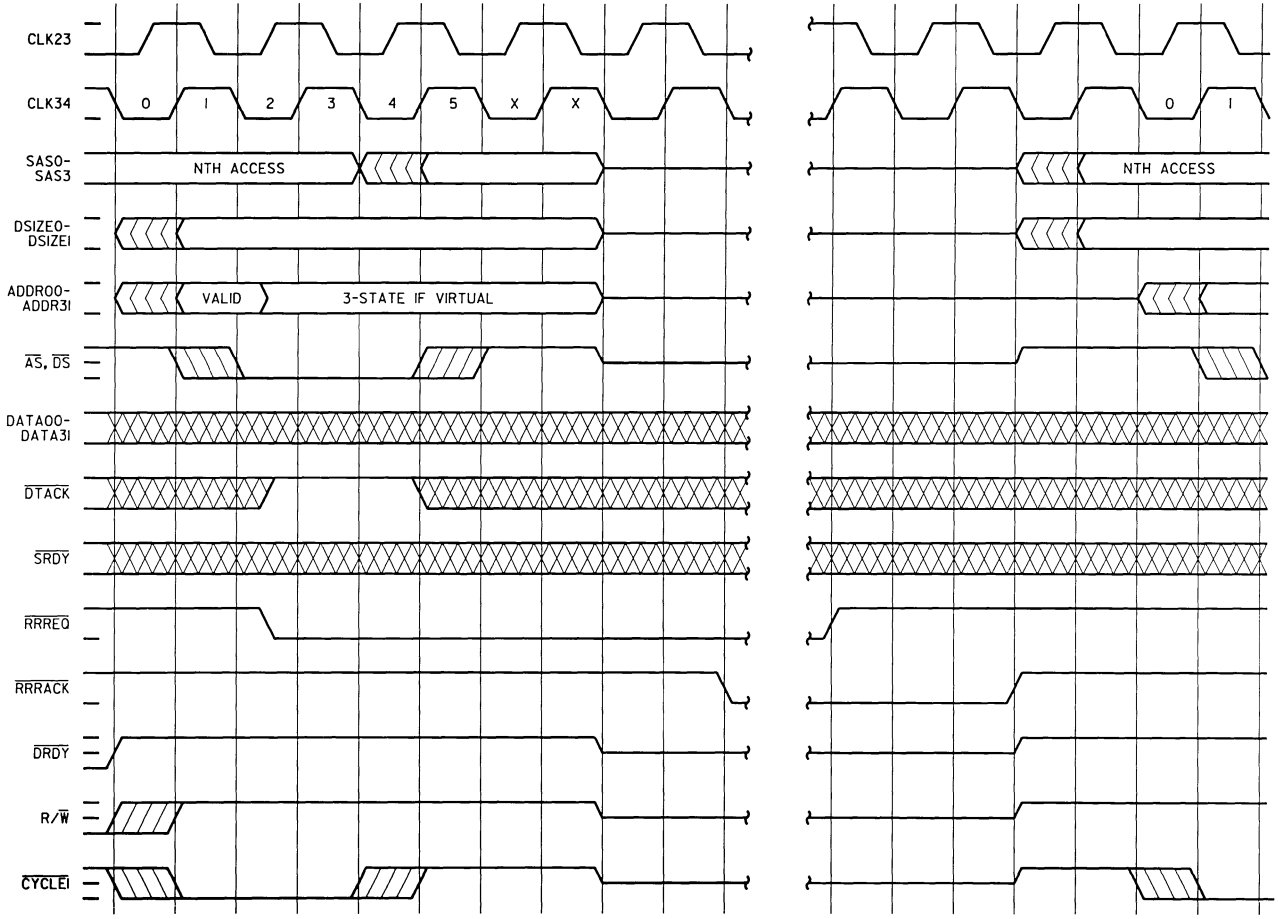


Figure 2-24. Relinquish and Retry

2.8 BLOCKFETCH SPECIAL CASES

As indicated in the descriptions of the bus exceptions, a fault, retry, or relinquish and retry of a blockfetch transaction is a special case of bus exception. The following descriptions address these special cases.

2.8.1 Fault on First Word of Blockfetch With Status Code Other Than Prefetch

If the CPU samples $\overline{\text{FAULT}}$ on the first word of a blockfetch where the SAS code is "instruction fetch" or "instruction fetch after PC discontinuity," the blockfetch transaction is altered as on Figure 2-25. The CPU sampled $\overline{\text{FAULT}}$ in clock state two and $\overline{\text{BLKFTCH}}$ in clock state three. Note that to sample $\overline{\text{BLKFTCH}}$, the CPU needs a $\overline{\text{DTACK}}$, $\overline{\text{SRDY}}$, or a bus exception. Upon seeing blockfetch and fault, the CPU removes $\overline{\text{DS}}$ at clock state five. It holds $\overline{\text{AS}}$ low and continues to hold it until the end of the first clock state X. The address bus, if in physical mode, is driven until the second clock state X. $\overline{\text{DRDY}}$ is not issued at all during this transaction.

2.8.2 Fault on First Word of Blockfetch With Status of Prefetch

As in other prefetch transactions, when the CPU is faulted, it ignores the data and just continues on with its current execution. This is illustrated on Figure 2-26. On clock state two, the CPU sampled $\overline{\text{FAULT}}$ and on clock state three, it sampled $\overline{\text{BLKFTCH}}$. The CPU terminated the first word fetch by removing $\overline{\text{DS}}$ (not issuing $\overline{\text{DRDY}}$), and continuing on to the second transaction. The second transaction operates normally.

2.8.3 Retry on First Word of Blockfetch

The CPU samples $\overline{\text{RETRY}}$ during clock state three and the $\overline{\text{BLKFTCH}}$ during clock state four. The CPU then terminates the transaction by removing $\overline{\text{DS}}$ at clock rate state five and the AS at the second clock state X. At this point the CPU waits for $\overline{\text{RETRY}}$ to go away. When it does the CPU retries the entire blockfetch transaction. This process is illustrated on Figure 2-27.

2.8.4 Retry on Second Word of Blockfetch

In this case (see Figure 2-28), the CPU sampled $\overline{\text{BLKFTCH}}$ and $\overline{\text{DTACK}}$, clocked the data from the data bus, issued a $\overline{\text{DRDY}}$, and continued on to the second word. During the first clock state W, the CPU samples $\overline{\text{RETRY}}$. It then terminates the transaction, does not issue a $\overline{\text{DRDY}}$, and waits for the $\overline{\text{RETRY}}$ to be removed. Since the second word of a blockfetch is always a prefetch, the CPU faults this transaction internally rather than retrying the entire blockfetch transaction. When the $\overline{\text{RETRY}}$ signal is removed, the CPU continues on with its current execution. If the CPU wants to do a new bus transaction it will proceed with this one since it will not retry the blockfetch.

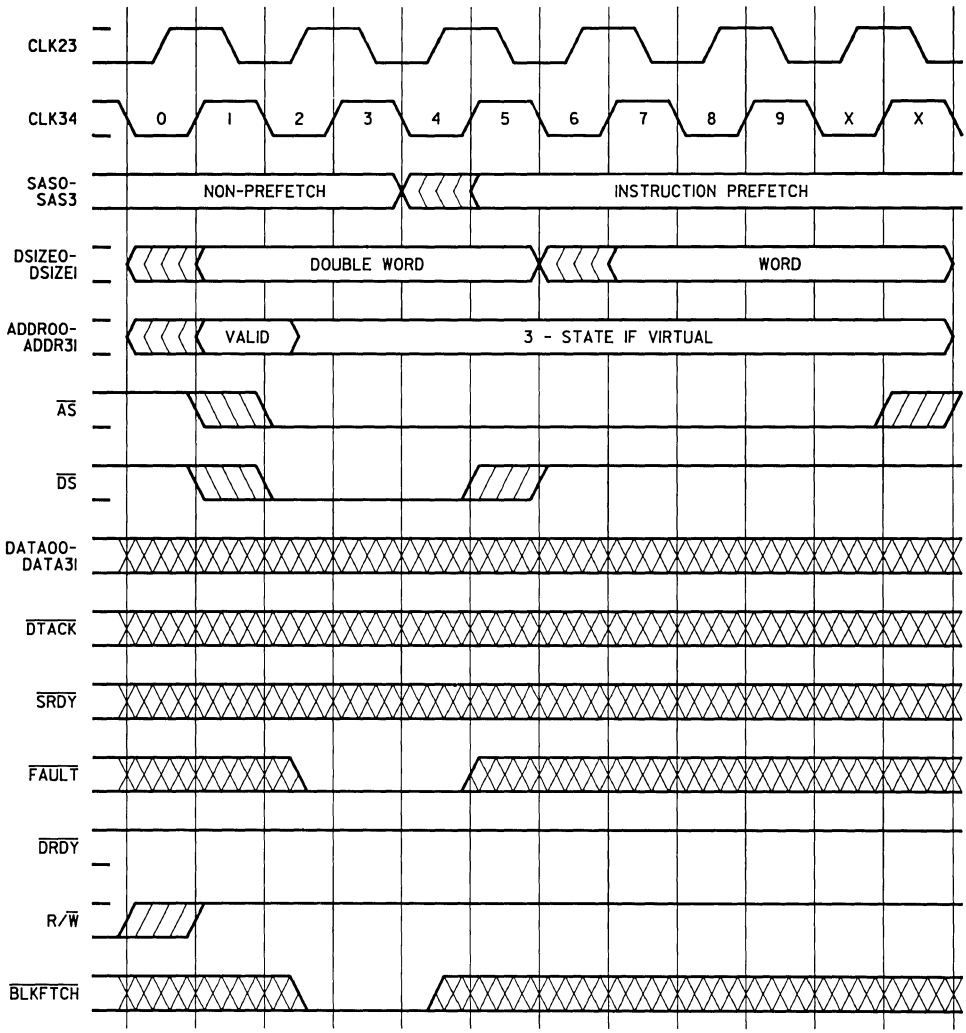


Figure 2-25. Fault on First Word of Blockfetch Transaction With Access Status Code (Not Instruction Prefetch)

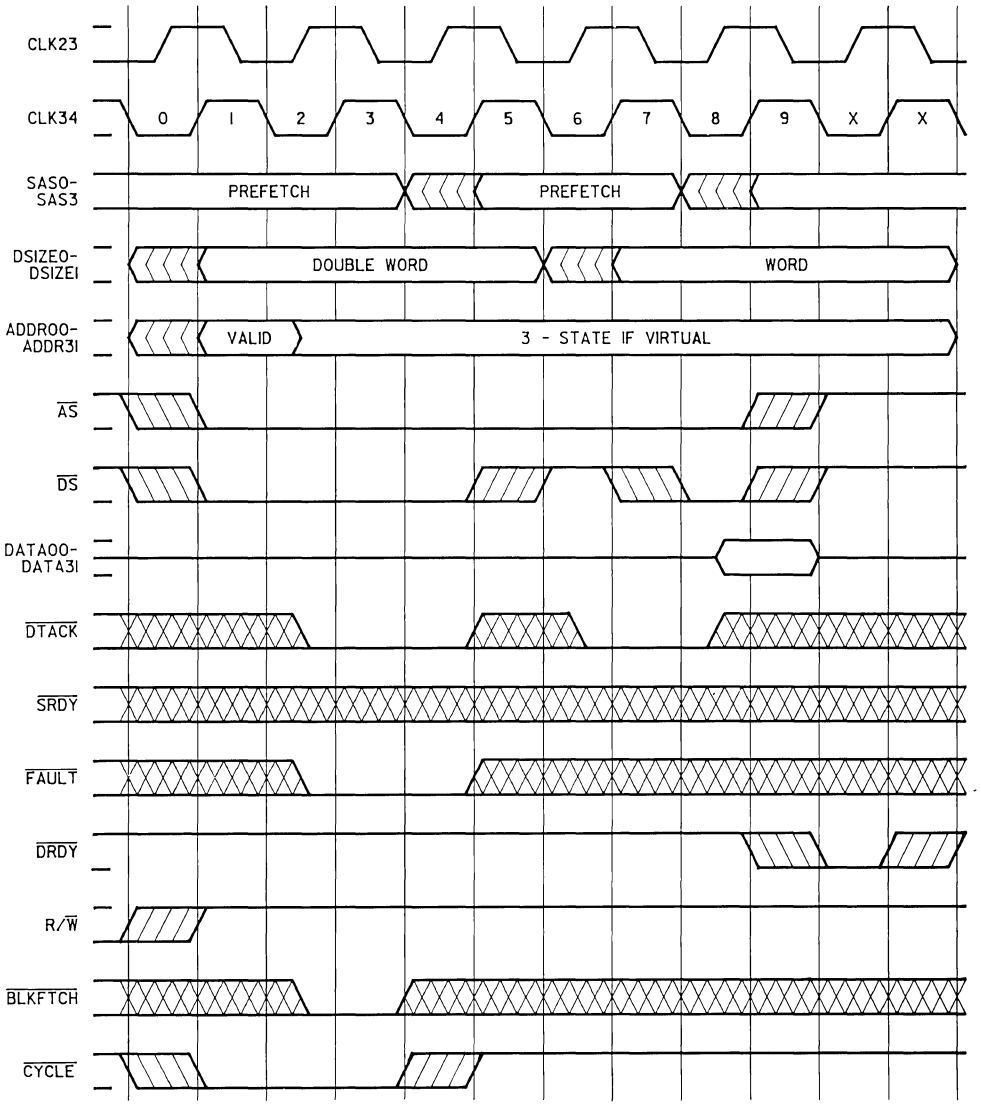
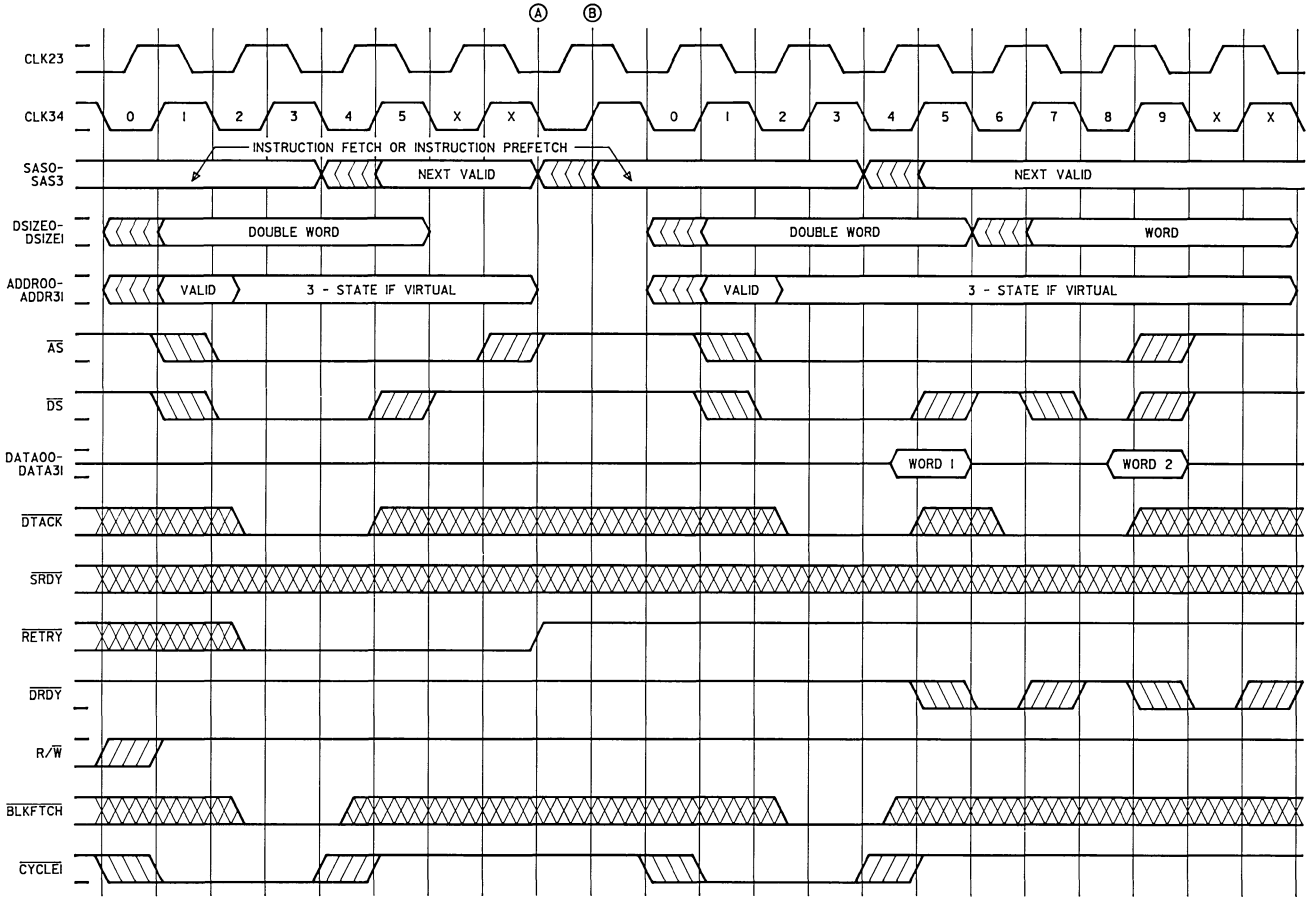
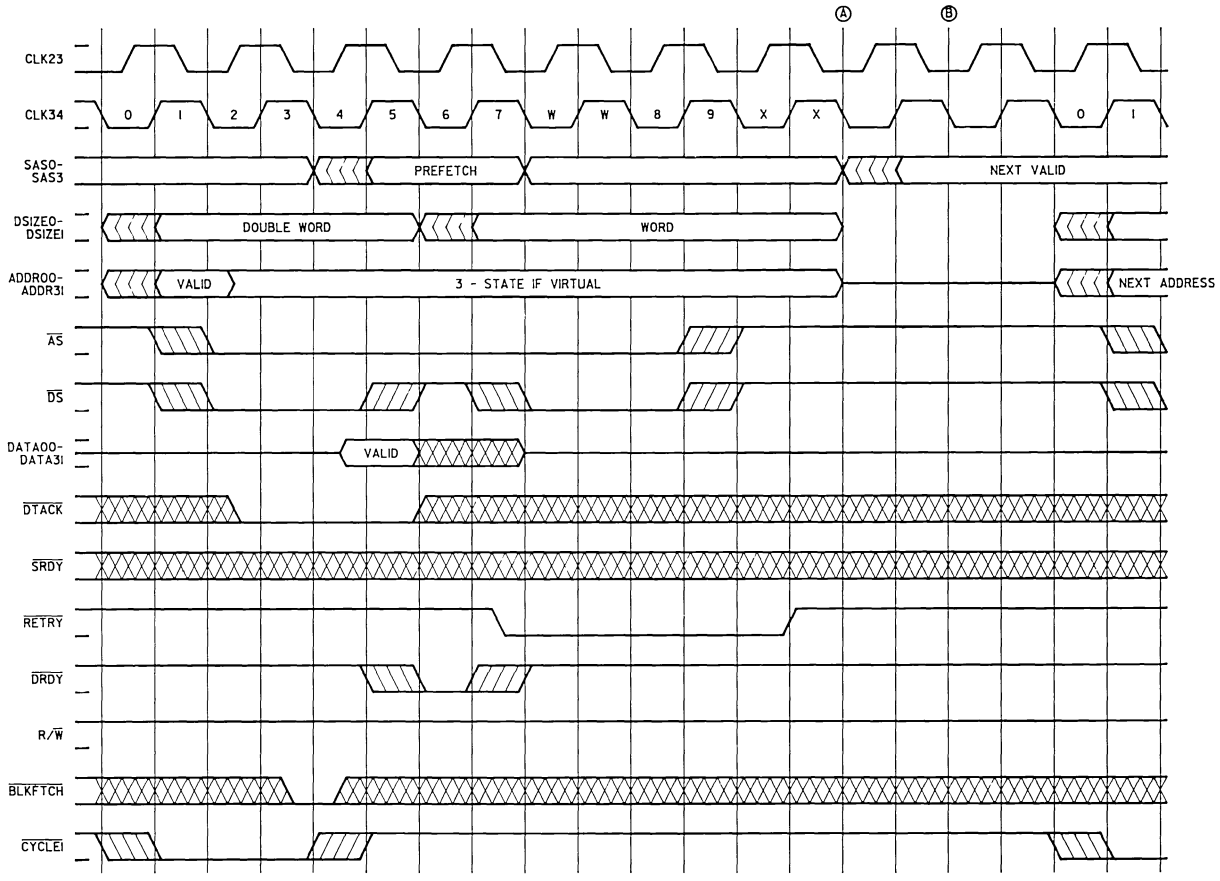


Figure 2-26. Fault on First Word of Blockfetch Transaction With Access Status Code of Prefetch



Note: If $\overline{\text{RRREQ}}$ is asserted instead of $\overline{\text{RETRY}}$, the CPU 3-states the bus at A and asserts $\overline{\text{RRRACK}}$ one clock cycle later at B.

Figure 2-27. Retry on First Word of Blockfetch Transaction



Notes:

1. On RRREQ, the following pins are 3-stated between points A and B . ADDR00– ADDR31, DATA00–DATA32, AS, CYCLEI, DRDY, DS, DSIZE0–DSIZE1, R/W, SAS0–SAS3.
2. If RRREQ or RETRY is asserted during the first word, then the entire blockfetch access is retried.

Figure 2-28. Retry on Second Word of Blockfetch

2.8.5 Relinquish and Retry on Blockfetch

Figures 2-27 and 2-28 can be used to illustrate the $\overline{\text{RRREQ}}$ bus exception for the first and second word of a blockfetch.

The timing and bus transaction for Figure 2-27 will look the same if the bus exception is $\overline{\text{RRREQ}}$ rather than $\overline{\text{RETRY}}$. However, the CPU will release the bus before doing the retried transaction. Additionally, the CPU will 3-state the bus at the end of the second clock state X (indicated by A on the diagram). One cycle later it will issue a relinquish and retry request acknowledge ($\overline{\text{RRRACK}}$) to tell the requesting device that it can now use the bus. When $\overline{\text{RRREQ}}$ is removed, the CPU will continue with the retried transaction starting at point B.

The same explanation applies for a $\overline{\text{RRREQ}}$ on the second word of a blockfetch (Figure 2-28). As above, the CPU will 3-state the bus at point A and issue a $\overline{\text{RRRACK}}$. Once the $\overline{\text{RRREQ}}$ is removed, the CPU will continue with the next bus transaction starting at point B and would not retry the blockfetch.

2.9 INTERRUPTS

The microprocessor accepts fifteen levels of interrupts. An interrupt request is made to the microprocessor by placing an interrupt request value on the interrupt priority level pins (IPL0—IPL3) or by requesting a nonmaskable interrupt by asserting $\overline{\text{NMINT}}$. Pending interrupts are not acknowledged until the currently executing instructions are completed. The exceptions to this are multiply, divide, modulo, move block word, string copy, and string end instructions which abort upon a pending interrupt.

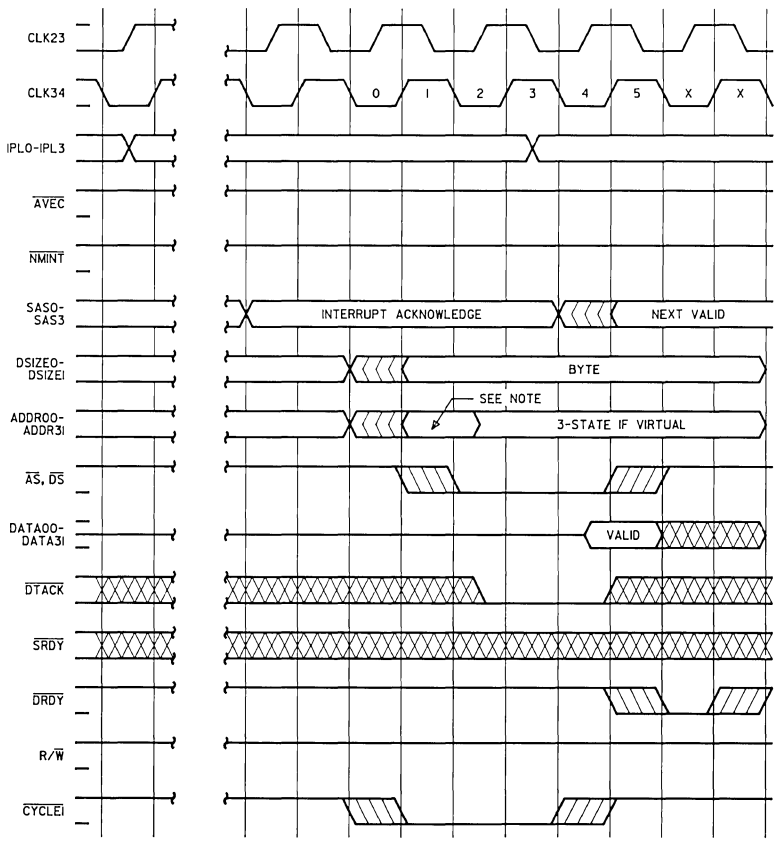
The pending interrupt value input on IPL0—IPL3 is internally inverted and compared to the value contained in the interrupt priority level (IPL) field of the processor status word (PSW). In order for the pending interrupt to be acknowledged, its inverted value must be greater than the IPL field value. Pending interrupts whose inverted values are equal to or less than the IPL field value are ignored. However, if the pending interrupt is nonmaskable, it will always interrupt the microprocessor regardless of the IPL field value.

The microprocessor also provides autovector, nonmaskable, and quick-interrupt facilities. The following describes these facilities.

2.9.1 Interrupt Acknowledge

The microprocessor acknowledges an interrupt by transmitting the inverted interrupt value on bits 2 through 5 of the address bus. In addition, the value placed on the interrupt option ($\overline{\text{INTOPT}}$) pin is inverted and transmitted on bit 6 of the address bus. (The $\overline{\text{INTOPT}}$ input has no effect on the microprocessor; however, it could be used to indicate, for example, whether the interrupt was hardware- or software-generated.) The microprocessor then fetches the interrupt vector number from the interrupting device on bits 0 through 7 of the data bus and begins execution of the interrupt handling routine. The interrupt acknowledge transaction is illustrated by Figure 2-29 which depicts the case where a value placed on the IPL0—IPL3 inputs causes an interrupt. In this case, the interrupt acknowledge is issued in response to the application of the IPL pins and $\overline{\text{INTOPT}}$

pin with no $\overline{\text{AVEC}}$ and $\overline{\text{NMINT}}$ active. During the interrupt acknowledge transaction, the CPU reads in an 8-bit offset provided by the interrupting device and used by the CPU as an offset to a table. The SAS code is "interrupt acknowledge." The DSIZE is a byte. The address corresponding to the interrupt acknowledge is indicated at the bottom of the figure. The interrupting device drives the data bus with the 8-bit offset and a memory acknowledge; in this case a $\overline{\text{DTACK}}$. The bus exceptions are accepted during this bus transaction. The IPL input values should be removed once the corresponding interrupt acknowledge has occurred.



Note: During the interrupt acknowledge the address bus (ADDR00—ADDR31) contains the following data.

31 ... 7	6	5	4	3	2	1	0
0 0	INVERTED $\overline{\text{INTOPT}}$ INPUT	INVERTED IPL3	IPL2	IPL1	IPL0	1	1

Figure 2-29. Interrupt Acknowledge

Table 2-4 summarizes how the interrupt priority levels are to be interpreted and shows the corresponding acknowledge for each level.

Table 2-4. Interrupt Level Code Assignments

Interrupt Request Input IPL0—IPL3 Bits: 3	2	1	0	Interrupt Option Input $\overline{\text{INTOPT}}$	Interrupt Acknowledge Output ADDR02—ADDR06 Bits: 06	05	04	03	02	Priority Level
0	0	0	0	0	1	1	1	1	1	Highest
0	0	0	0	1	0	1	1	1	1	Priority
0	0	0	1	0	1	1	1	1	0	2nd
0	0	0	1	1	0	1	1	1	0	
0	0	1	0	0	1	1	1	0	1	3rd
0	0	1	0	1	0	1	1	0	1	
0	0	1	1	0	1	1	1	0	0	4th
0	0	1	1	1	0	1	1	0	0	
0	1	0	0	0	1	1	0	1	1	5th
0	1	0	0	1	0	1	0	1	1	
0	1	0	1	0	1	1	0	1	0	6th
0	1	0	1	1	0	1	0	1	0	
0	1	1	0	0	1	1	0	0	1	7th
0	1	1	0	1	0	1	0	0	1	
0	1	1	1	0	1	1	0	0	0	8th
0	1	1	1	1	0	1	0	0	0	
1	0	0	0	0	1	0	1	1	1	9th
1	0	0	0	1	0	0	1	1	1	
1	0	0	1	0	1	0	1	1	0	10th
1	0	0	1	1	0	0	1	1	0	
1	0	1	0	0	1	0	1	0	1	11th
1	0	1	0	1	0	0	1	0	1	
1	0	1	1	0	1	0	1	0	0	12th
1	0	1	1	1	0	0	1	0	0	
1	1	0	0	0	1	0	0	1	1	13th
1	1	0	0	1	0	0	0	1	1	
1	1	0	1	0	1	0	0	1	0	14th
1	1	0	1	1	0	0	0	1	0	
1	1	1	0	0	1	0	0	0	1	Lowest
1	1	1	0	1	0	0	0	0	1	Priority
1	1	1	1	0	x	x	x	x	x	No Interrupt
1	1	1	1	1	x	x	x	x	x	Pending

x signifies no value placed on address bus.

2.9.2 Auto-vector Interrupt

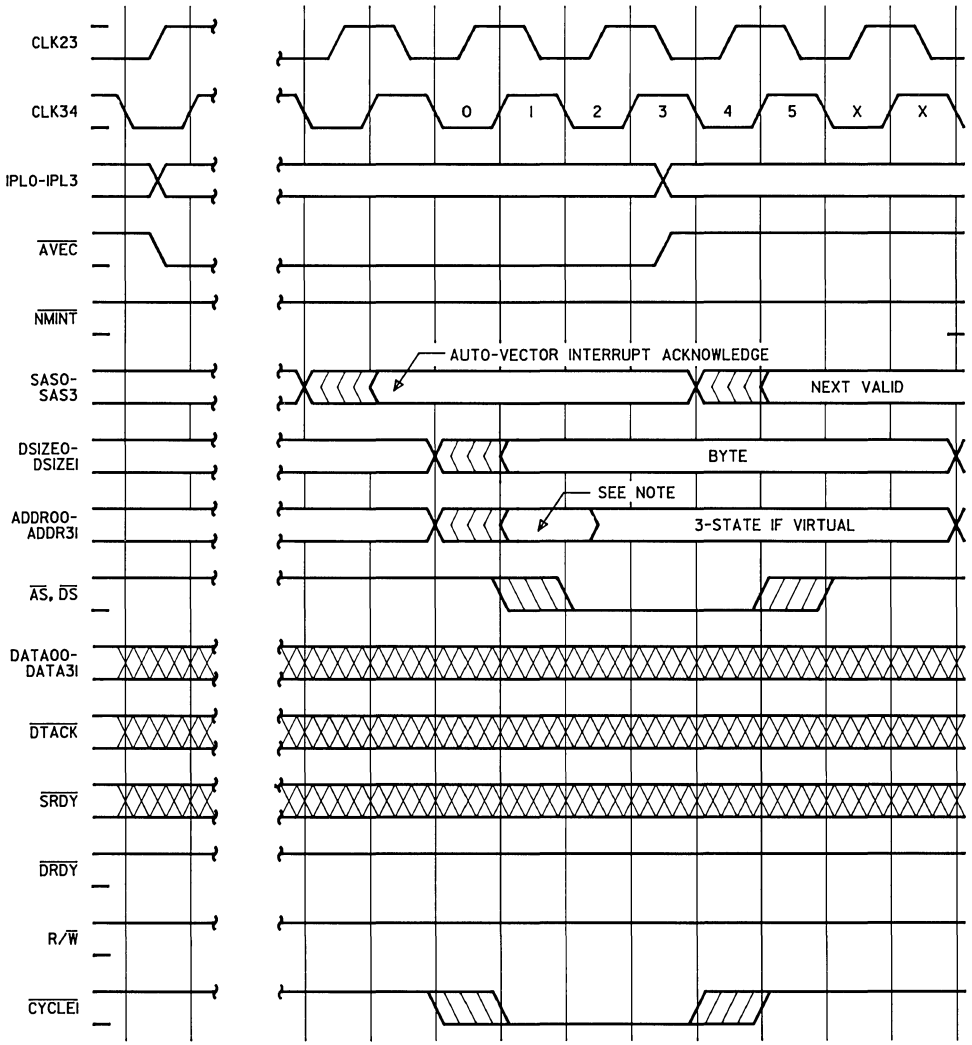
If the auto-vector (AVEC) input is active during an interrupt request, the microprocessor will not fetch a vector number from the interrupting device. Instead, the microprocessor provides the interrupt vector by treating the inverted INTOPT input, concatenated with the interrupt priority level input (IPL0—IPL3), as a vector number. The auto-vector facility reduces hardware costs in smaller, less complex systems because the interrupt vector is supplied by the microprocessor instead of by external hardware.

Refer to Figure 2-30 for an illustration of the auto-vector interrupt acknowledge transaction. In this transaction, an auto-vector acknowledge is issued in response to the application of the IPL pins and INTOPT pin with AVEC active and no NMINT. Since the CPU does not need to read in an external value, it does an auto-vector interrupt acknowledge without looking for a memory acknowledge or a bus exception. The transaction goes through the clock states without inserting wait cycles. This transaction is used to tell the interrupting device that it should remove the IPL and AVEC input values. No DRDY is issued because there is no latching of data.

2.9.3 Nonmaskable Interrupt

The nonmaskable interrupt facility is provided to satisfy reliability and recoverability requirements of various systems. As previously mentioned, a nonmaskable interrupt can interrupt the microprocessor regardless of the current priority level in the IPL field. A nonmaskable interrupt occurs if the nonmaskable interrupt input (NMINT) is asserted. The interrupt is then treated as an autovector interrupt with vector number 0. During the interrupt acknowledge cycle of a nonmaskable interrupt, address bus bits ADDR00—ADDR31 contain zeros. This distinguishes a nonmaskable interrupt from all other interrupts.

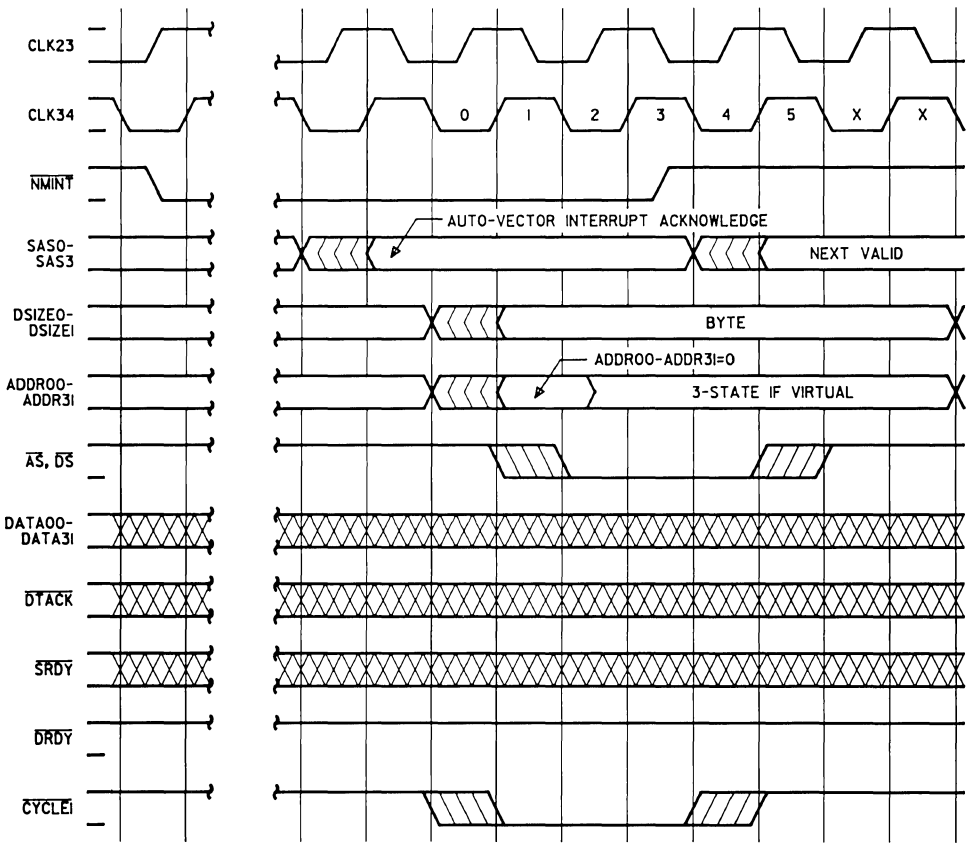
Figure 2-31 illustrates the nonmaskable interrupt acknowledge transaction. Here, a nonmaskable interrupt acknowledge is issued in response to the application of the NMINT input. For a nonmaskable interrupt, the CPU uses an internal offset corresponding to an IPL of zero. Since the CPU does not need to read in data, it performs the transaction without looking for a memory acknowledge or a bus exception. The transaction goes through the clock states without inserting wait cycles. Again, the interrupting device should release NMINT when it sees the acknowledge. The SAS code is "auto-vector acknowledge," but the interrupt vector is 0. ADDR00 can be used to determine the difference between the AVEC and NMINT interrupts. It is a 1 for auto-vector and a 0 for nonmaskable interrupt.



Note: During the interrupt acknowledge the address bus (ADDR00—ADDR31) contains the following data.

31 ... 7	6	5	4	3	2	1	0
0 0	INVERTED INTOPT INPUT	INVERTED IPL3	IPL2	IPL1	IPL0	1	1

Figure 2-30. Auto-vector Interrupt Acknowledge



Note: The address bus ADDR00—ADDR31 contains all zeroes during the acknowledge of a nonmaskable interrupt.

Figure 2-31. Nonmaskable Interrupt Acknowledge

2.9.4 Quick Interrupt

The quick-interrupt facility enchances the performance of systems that do not require the functionality of the "full interrupt." Its handling routine (a microsequence that stores the PSW and PC) requires less time than that of a "full interrupt." All interrupts are serviced via the quick-interrupt facility if the quick-interrupt enable (QIE) bit in the PSW is set (1). Table 2-5 summarizes how the microprocessor handles the various interrupt requests. See Chapter 4 for more information on full and quick interrupts.

2.10 BUS ARBITRATION

The microprocessor's bus may be requested in two ways. External devices may request the bus by asserting the relinquish and retry request input ($\overline{\text{RRREQ}}$), as explained previously, or by asserting the bus request input ($\overline{\text{BUSREQ}}$).

The relinquish and retry request has priority over a bus request. The microprocessor will only acknowledge a relinquish and retry request during bus transactions; however, it will ignore the request during the write portion of a read interlocked transaction.

A bus request during a CPU bus transaction is not acknowledged until the end of a bus transaction or until the end of the write portion of a read interlocked transaction.

2.10.1 Bus Request During a Bus Transaction

$\overline{\text{BUSRQ}}$ is sampled independently of bus transactions at the beginning of every clock cycle. On Figure 2-32 it is sampled for the first time at the beginning of clock state two. After sampling $\overline{\text{BUSRQ}}$, the CPU continues the current bus transaction. After the transaction is completed, the CPU 3-states the address and data buses and some control signals just after the last clock state X. A cycle later it issues the bus request acknowledge, $\overline{\text{BRACK}}$. At this point the device requesting the bus can perform its operations. When finished, the device drops the $\overline{\text{BUSRQ}}$. After seeing this drop, the CPU removes $\overline{\text{BRACK}}$ and takes back the bus. Note that if the bus request occurred during an active retry request or relinquish and retry request it would not be acknowledged until after the current transaction had been retried. Refer to **2.19 SUPPLEMENTARY PROTOCOL DIAGRAMS** for an example.

For a bus request that does not occur during a bus transaction, the CPU will 3-state the bus a cycle after sampling $\overline{\text{BUSRQ}}$ and issue $\overline{\text{BRACK}}$ a cycle after that.

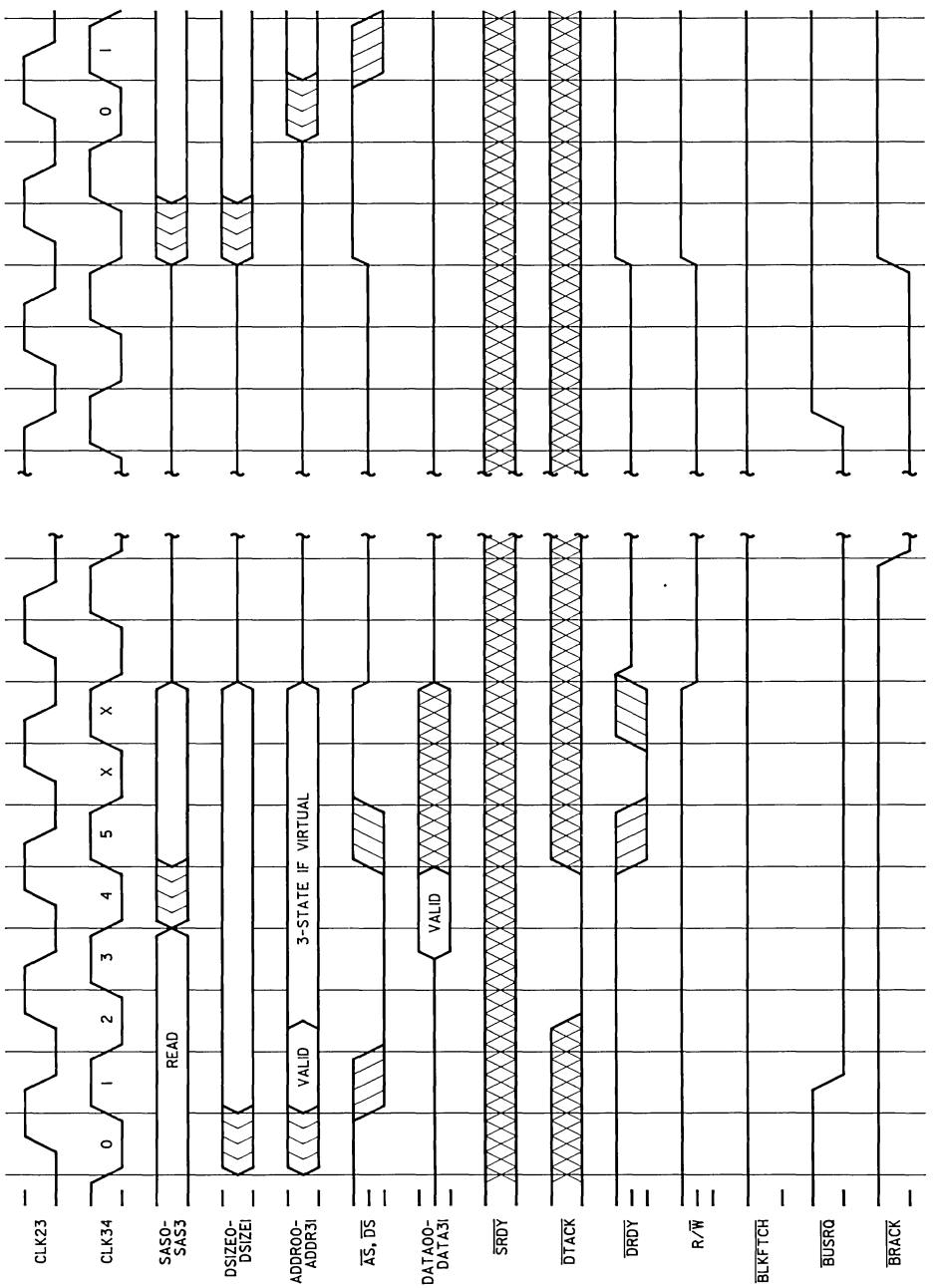


Figure 2-32. Bus Request During a Transaction

Table 2-5. Interrupt Acknowledge Summary

Interrupt Priority	Interrupt Acknowledge	$\overline{\text{AVEC}}$	$\overline{\text{NMINT}}$	QIE	Result
Less than PSW IPL field priority	No	x	1	x	Interrupt is not acknowledged.
Equal to PSW IPL field priority	No	x	1	x	Interrupt is not acknowledged.
Greater than PSW IPL field priority	Yes	1	1	0	Interrupt is acknowledged and serviced via the full-interrupt sequence. Microprocessor fetches vector number from interrupting device.
Greater than PSW IPL field priority	Yes	0	1	0	Interrupt is acknowledged and serviced via the full-interrupt sequence. Microprocessor supplies the vector number.
Any level compared to PSW IPL field priority	Yes	x	0	0	Interrupt is acknowledged and serviced via the full-interrupt sequence. It is treated as an auto-vector at vector number 0. The address bus contains all zeros during the acknowledge.
Greater than PSW IPL field priority	Yes	1	1	1	Interrupt is acknowledged and serviced via quick-interrupt sequence. Microprocessor fetches vector number from interrupting device.
Greater than PSW IPL field priority	Yes	0	1	1	Interrupt is acknowledged and serviced via quick-interrupt sequence. Microprocessor supplies the vector number.
Any level compared to PSW IPL field priority	Yes	x	0	1	Interrupt is acknowledged and serviced via quick-interrupt sequence. It is treated as an auto-vector interrupt at vector number 0. The address bus contains all zeros during the acknowledge.

2.10.2 DMA Operation

The microprocessor provides the support for direct memory access (DMA) and shares bus control responsibilities with the system DMA controller. To initiate a DMA operation, the controller requests the microprocessor bus by asserting (BUSRQ). Recall that this request is not acknowledged until the end of a bus transaction or until the end of the write portion of a read interlocked transaction. However, if the CPU is not using the bus, the request is acknowledged immediately. Once the microprocessor recognizes the request, it 3-states the following signals:

$\overline{\text{ABORT}}$	DATA00—DATA31	R/$\overline{\text{W}}$
ADDR00—ADDR31	$\overline{\text{DRDY}}$	SAS0—SAS3
$\overline{\text{AS}}$	$\overline{\text{DS}}$	$\overline{\text{VAD}}$
$\overline{\text{CYCLEI}}$	DSIZE0—DSIZE1	XMD0—XMD1

After the microprocessor has 3-stated the above signals, it acknowledges the DMA request by asserting the bus request acknowledge output ($\overline{\text{BRACK}}$). Table 2-6 summarizes the output signal states once the DMA has been acknowledged.

Terminating a DMA operation reverses the start of DMA. The DMA controller removes the request by negating $\overline{\text{BUSRQ}}$ (drives the input high). The microprocessor then negates the acknowledge ($\overline{\text{BRACK}}$), and, finally, the 3-stated signals are returned to the microprocessor's control. The next operation may then begin.

Table 2-6. Output Signal States After DMA Request is Acknowledged

Output Signal	Signal State	Output Signal	Signal State
$\overline{\text{ABORT}}$	Z'	DSIZE0—DSIZE1	Z
ADDR00—ADDR31	Z	R/$\overline{\text{W}}$	Z'
$\overline{\text{AS}}$	Z'	$\overline{\text{RESET}}$	Logic 1
$\overline{\text{BRACK}}$	Logic 0	$\overline{\text{RRRACK}}$	Logic 1
$\overline{\text{CYCLEI}}$	Z'	SAS0—SAS3	Z'
DATA00—DATA31	Z	$\overline{\text{VAD}}$	Z
$\overline{\text{DRDY}}$	Z'	XMD0—XMD1	Z
$\overline{\text{DS}}$	Z'		

Where:
Z High impedance state.
Z' High impedance. Held at logic 1 with external passive hold resistor.

2.11 RESET

The microprocessor handles two types of reset requests: system and internal. A reset has the highest priority and will preempt any ongoing microprocessor operation.

2.11.1 System Reset

A system reset is initiated when the system drives the reset request input ($\overline{\text{RESETR}}$) low. This double-latched input must be active on three consecutive latchings before being recognized. This ensures noise immunity. After recognizing the reset request, the microprocessor sends a reset acknowledge to the system by asserting $\overline{\text{RESET}}$. All microprocessor outputs are then driven to a temporary state that prevents control signal and bus conflicts while the system responds to the reset acknowledge.

Once the system has responded to the acknowledge, it negates $\overline{\text{RESETR}}$. The microprocessor continues to hold $\overline{\text{RESET}}$ active for 128 clock cycles after $\overline{\text{RESETR}}$ has been negated, allowing the external system to go through its own initialization sequence. At the end of this period the microprocessor negates $\overline{\text{RESET}}$ and begins executing the internal reset sequence. Table 2-6 indicates the states of the microprocessor's output pins once $\overline{\text{RESET}}$ is negated. During this sequence, the microprocessor performs the following register initialization to restart the operation.

- The microprocessor changes to physical addressing mode.
- The microprocessor fetches a word at location 80 hexadecimal and stores it in the process control block pointer (PCBP). This word is the beginning address of the reset process control block, PCB.
- The microprocessor fetches a word at the PCB address and stores it in the processor status word.
- The microprocessor fetches a word at the location four bytes from the PCB address and stores it in the program counter (PC). This word is the PC value for initial execution.
- The microprocessor fetches a word at the location eight bytes from the initial PCB address and stores it in the stack pointer.
- If the PSW I bit is set (1), the microprocessor clears the bit (0), fetches a word at the location twelve bytes from the initial PCBP, and stores it as the new PCBP.
- The microprocessor begins execution at the address specified by the PC.

2.11.2 Internal Reset

An internal reset sequence is like a system reset sequence except there is no external reset request signal. The request is generated internally. Note that the $\overline{\text{RESET}}$ line will still go active for 128 clock cycles after $\overline{\text{RESETR}}$ is released.

Table 2-7. Output States on Reset

Output Signal	Signal State	
	CPU* is Bus Arbiter	CPU* is Not Bus Arbiter
$\overline{\text{ABORT}}$	Logic 1	High Impedance
ADDR00—ADDR31	High Impedance	High Impedance
$\overline{\text{AS}}$	Logic 1	High Impedance
$\overline{\text{BRACK}}$	Logic 1	—
$\overline{\text{BUSRQ}}$	—	Logic 1
$\overline{\text{CYCLEI}}$	Logic 1	High Impedance
DATA00—DATA31	High Impedance	High Impedance
$\overline{\text{DRDY}}$	Logic 1	High Impedance
$\overline{\text{DS}}$	Logic 1	High Impedance
DSIZE0, DSIZE1	Logic 0	High Impedance
IQS0, IQS1	Logic 1	Logic 1
R/$\overline{\text{W}}$	Logic 1	High Impedance
$\overline{\text{RRRACK}}$	High Impedance, (a)	High Impedance, (a)
SAS0—SAS3	Logic 1	High Impedance
$\overline{\text{SOI}}$	Logic 1	Logic 1
$\overline{\text{VAD}}$	(b)	High Impedance
XMD0, XMD1	(c)	High Impedance

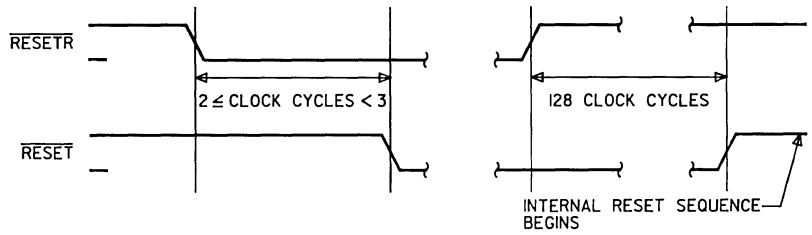
*CPU is the *WE* 32100 Microprocessor.

Notes:

a Open drain output not actively driven under this condition.

b Not guaranteed to be logic 1 (i.e., physical address) until approximately 38 clock cycles after $\overline{\text{RESET}}$ is negated.

c Not guaranteed to be in kernel mode until approximately 18 clock cycles after $\overline{\text{RESET}}$ is negated.



Note: RESETR must be asserted for at least two clock cycles to be recognized. RESET is negated 128 clock cycles after negation of RESETR.

Figure 2-33. Reset Sequence

2.11.3 Reset Sequence

The reset sequence is depicted on Figure 2-33. As previously stated, after RESETR is sampled for at least two consecutive clock cycles, the CPU issues the reset acknowledge (RESET). While RESETR is active, the CPU holds RESET active. Once RESETR is removed by the requesting device, the CPU counts 128 clock cycles and then removes RESET. At this point the CPU enters the internal reset sequence (see Chapter 4).

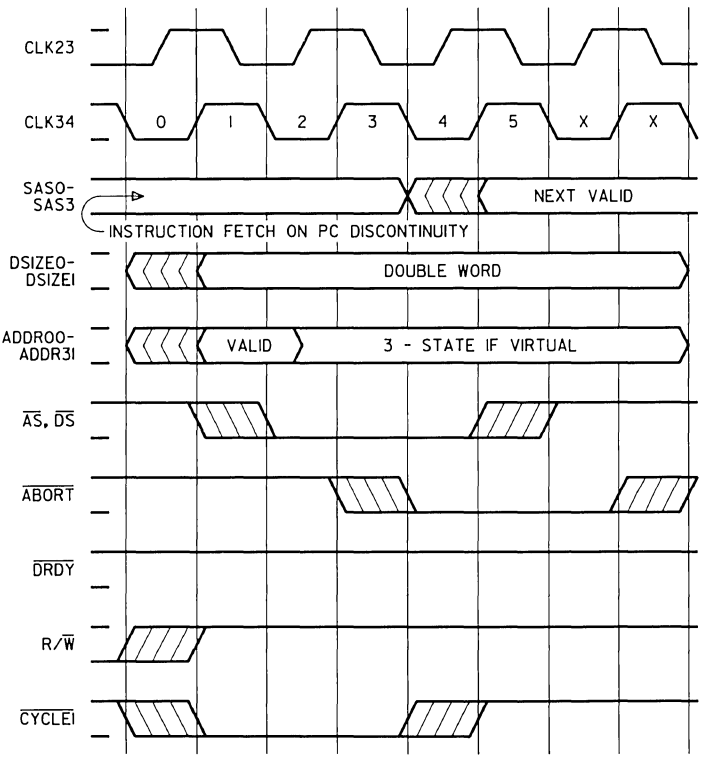
Note that if the CPU receives a fault during certain high-level bus transactions it can enter a reset exception (see **2.15 EXCEPTIONAL CONDITIONS**). This exception goes through a simulated system reset and includes issuing RESET for 128 clock cycles.

2.12 ABORTED MEMORY ACCESSES

There are two events that cause the CPU to abort a memory access; when the CPU has a program counter (PC) discontinuity with an instruction cache hit, and when an alignment fault occurs. These two events are illustrated next.

2.12.1 Aborted Access on PC Discontinuity With Instruction Cache Hit

Figure 2-34 depicts the protocol associated with this event. When the CPU does a PC discontinuity it starts to fetch the next instruction word from memory. The SAS code is "instruction fetch after PC discontinuity." If there is a hit in the cache for this instruction fetch, the CPU cancels the external instruction fetch by terminating the transaction. The CPU ignores memory acknowledges and bus exceptions during this transaction. To indicate that it is terminating the transaction, the CPU issues $\overline{\text{ABORT}}$ for two cycles, starting with clock state four. No $\overline{\text{DRDY}}$ is issued and the CPU ignores the data bus. The CPU uses the instruction word that it obtained from the instruction cache.

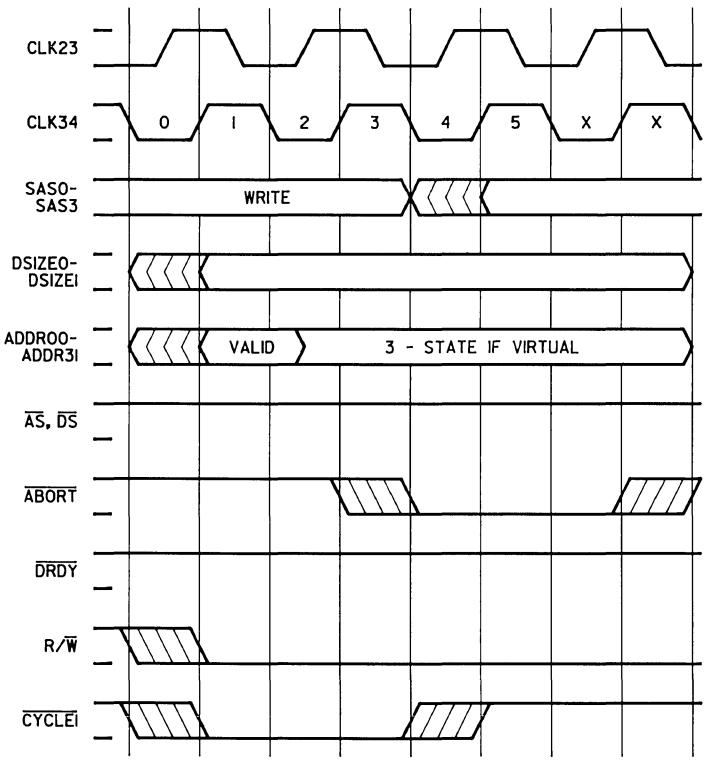


Note: $\overline{\text{BLKFTCH}}$, DATA00—DATA31, $\overline{\text{DTACK}}$, $\overline{\text{FAULT}}$, $\overline{\text{RETRY}}$, $\overline{\text{RRREQ}}$, and $\overline{\text{SRDY}}$ are ignored.

Figure 2-34. Aborted Access on I-Cache Hit With PC Discontinuity

2.12.2 Alignment Fault Bus Activity

If the CPU detects an alignment fault on an intended CPU-generated bus transaction, it will terminate the transaction and proceed to the fault handler. The write transaction on Figure 2-35 started with the address bus, as well as the DSIZE, SAS, R/$\overline{W}$, and $\overline{CYCLEI}$ being driven by the CPU. The CPU detects the alignment fault and does not issue $\overline{AS}$ and $\overline{DS}$. It issues $\overline{ABORT}$, starting at clock state three, to indicate that it is terminating the transaction. The CPU ignores memory acknowledges and bus exceptions during this time (see note Figure 2-35). $\overline{DRDY}$ is not issued.



Notes:

1. DATA00—DATA31, $\overline{DTACK}$, $\overline{FAULT}$, $\overline{RETRY}$, $\overline{RRREQ}$, and $\overline{SRDY}$ are ignored.
2. Protocol is the same for a read transaction.

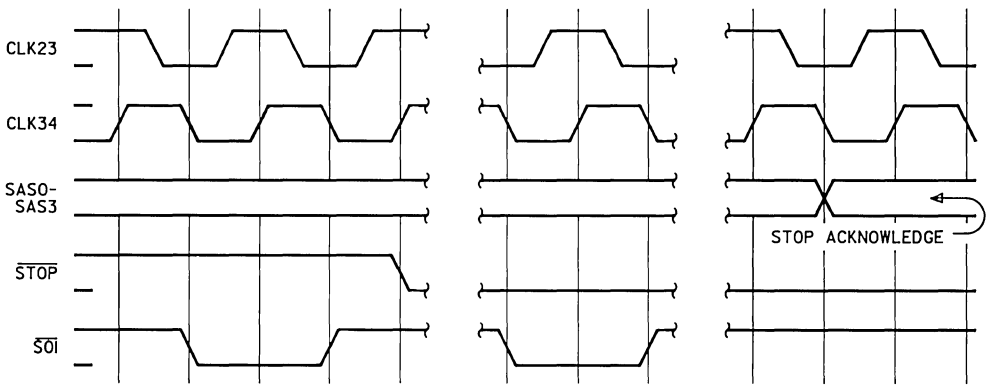
Figure 2-35. Alignment Fault Bus Activity (Write Transaction Is Shown)

2.13 SINGLE-STEP OPERATION

Hardware single-step can be performed by use of the stop input ($\overline{STOP}$). This input halts the execution of instructions beyond the ones already started by the microprocessor. Because of the pipelined architecture, the CPU may execute, at most, one more instruction beyond the instruction during which $\overline{STOP}$ was asserted. The microprocessor then remains in a halt state until the $\overline{STOP}$ input is released.

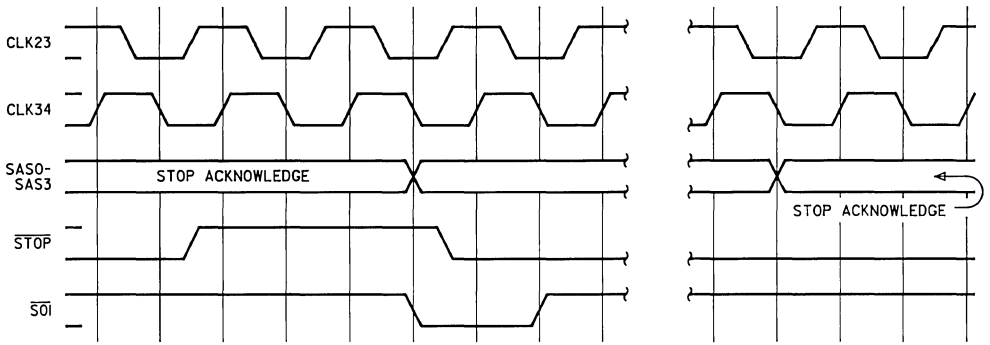
A bus request ($\overline{BUSREQ}$) is honored while the microprocessor is halted. Additionally, interrupts are acknowledged upon release of $\overline{STOP}$, but not while $\overline{STOP}$ remains asserted.

Figure 2-36 depicts the start of single-step operation. The operation is started by the assertion of $\overline{STOP}$. The CPU will complete the current instruction and execute, at most, one more instruction. After this the CPU stops execution and issues the SAS code "stop acknowledge." The CPU will remain in this state until $\overline{STOP}$ is released.



Notes:
1. At most, one full assertion of $\overline{SOI}$ may appear before $\overline{STOP}$ is acknowledge.
2. $\overline{BARB}$ = 0 and $\overline{BRACK}$ = 1 in order to see stop acknowledge access status code.

Figure 2-36. Start of Single-Step Operation



Note: $\overline{BARB}$ = 0 and $\overline{BRACK}$ = 1 in order to see stop acknowledge access status code.

Figure 2-37. Single-Step Operation

After the CPU has stopped, and until a start of instruction output ($\overline{SOI}$) is issued, instruction by instruction execution can be performed by releasing $\overline{STOP}$. At this point, immediate application and holding of $\overline{STOP}$ will prevent a second instruction from starting. With $\overline{STOP}$ asserted, the CPU will complete the instruction and issue the stop acknowledge SAS code. To resume normal execution, $\overline{STOP}$ must be completely released. The single-step operation is shown on Figure 2-37.

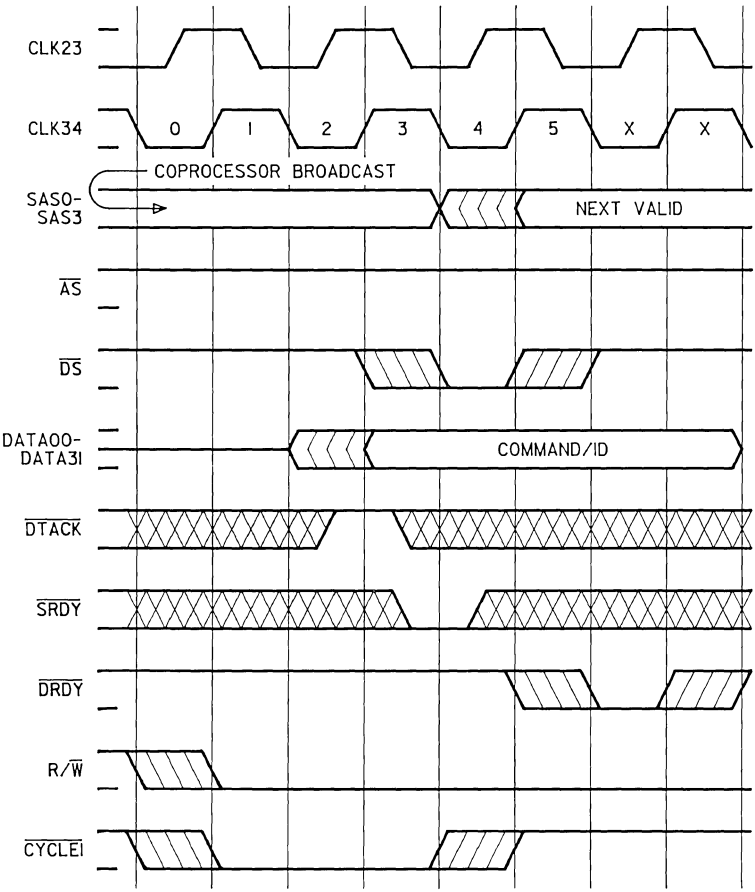
2.14 COPROCESSOR OPERATIONS

The *WE* 32100 Microprocessor provides a coprocessor interface consisting of ten instructions and the associated pinout and bus transactions. The coprocessor interface assures high performance and system throughput. When a coprocessor instruction is executed by the CPU, a series of bus transactions occur. The following details the process and provides the associated protocol.

2.14.1 Coprocessor Broadcast

This transaction notifies the coprocessor of the action the CPU wants performed. To prevent memory from being selected, $\overline{AS}$ is not issued during this transaction. Since this is a write operation, R/$\overline{W}$ is in write mode and the timing of $\overline{DS}$ is for a write. The CPU drives the data bus with the information that it wants to send to the coprocessor. The coprocessor responds with a memory acknowledge. The CPU then terminates the transaction and goes on to the next one. The CPU will insert up to two wait cycles while it

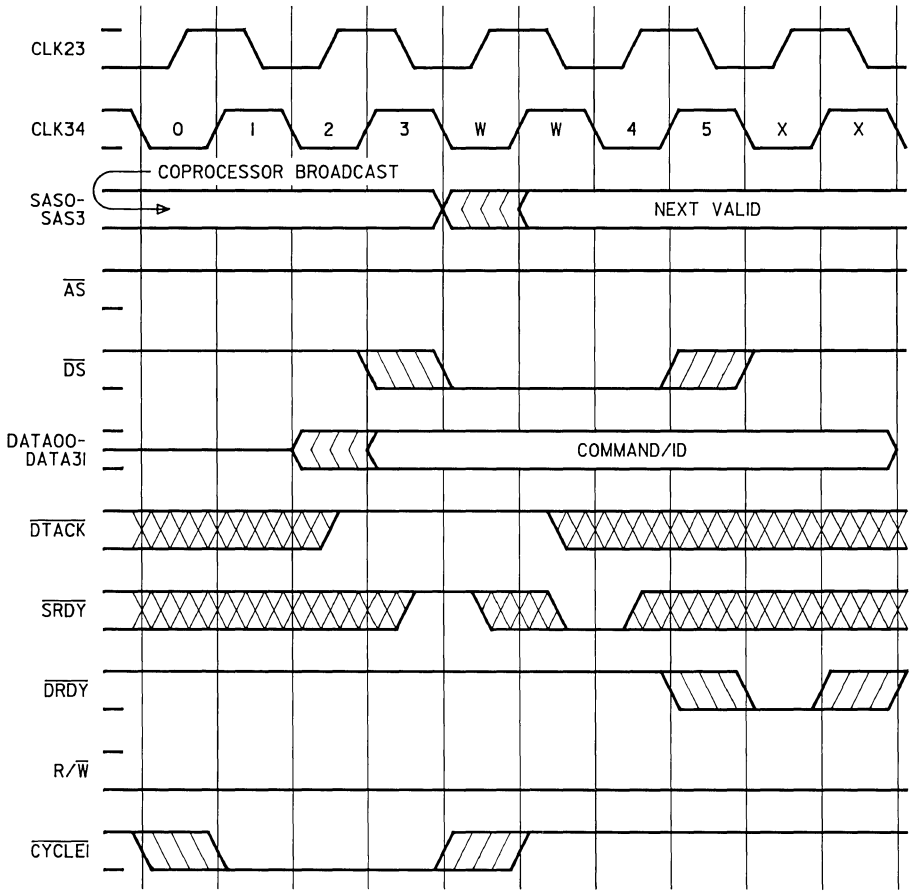
waits for the memory acknowledge from the coprocessor. This gives the coprocessor a limited time to respond to this transaction. Figure 2-38 shows the zero, one, and two wait cycle cases before the coprocessor responds with a memory acknowledge (in this case, $\overline{SRDY}$).



A. Zero Wait Cycles

Notes:
1. Zero, one, and two wait cycles using $\overline{SRDY}$.
2. Greater than two wait cycles causes internal CPU memory fault.

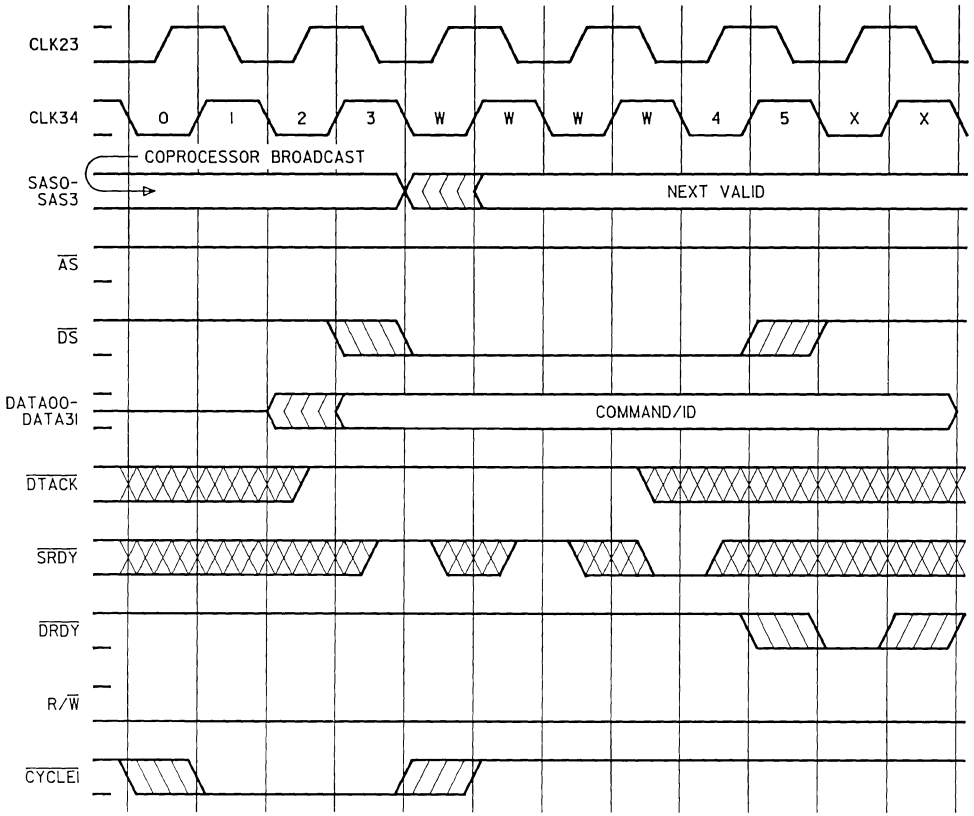
Figure 2-38. Coprocessor Command and ID Transfer (Sheet 1 of 3)



B. One Wait Cycle

Notes:
1. Zero, one, and two wait cycles using SRDY.
2. Greater than two wait cycles causes internal CPU memory fault.

Figure 2-38. Coprocessor Command and ID Transfer (Sheet 2 of 3)



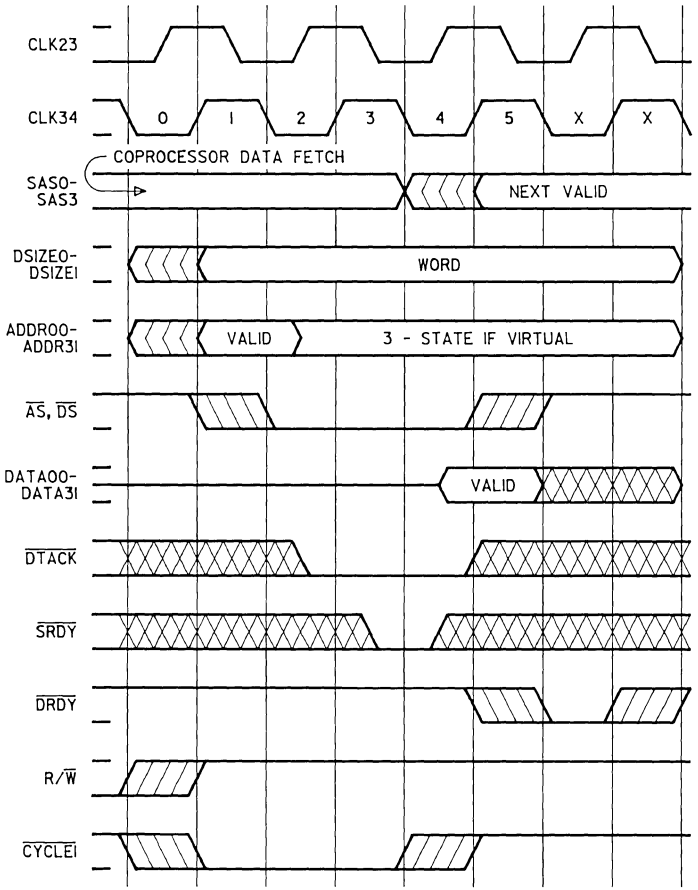
C. Two Wait Cycles

Notes:
1. Zero, one, and two wait cycles using SRDY.
2. Greater than two wait cycles causes internal CPU memory fault.

Figure 2-38. Coprocessor Command and ID Transfer (Sheet 3 of 3)

2.14.2 Coprocessor Operand Fetch

After doing a broadcast, the CPU will perform from zero to three coprocessor operand fetch transactions, depending on which coprocessor instruction is being executed. For this transaction, the CPU goes through the motions of doing a read from the memory, but the coprocessor latches the data as it sees it on the bus. The SAS is "coprocessor data fetch," and DSIZE is a word. The memory is issuing the acknowledge for this transaction. Figure 2-40 shows the protocol for a single coprocessor operand fetch.

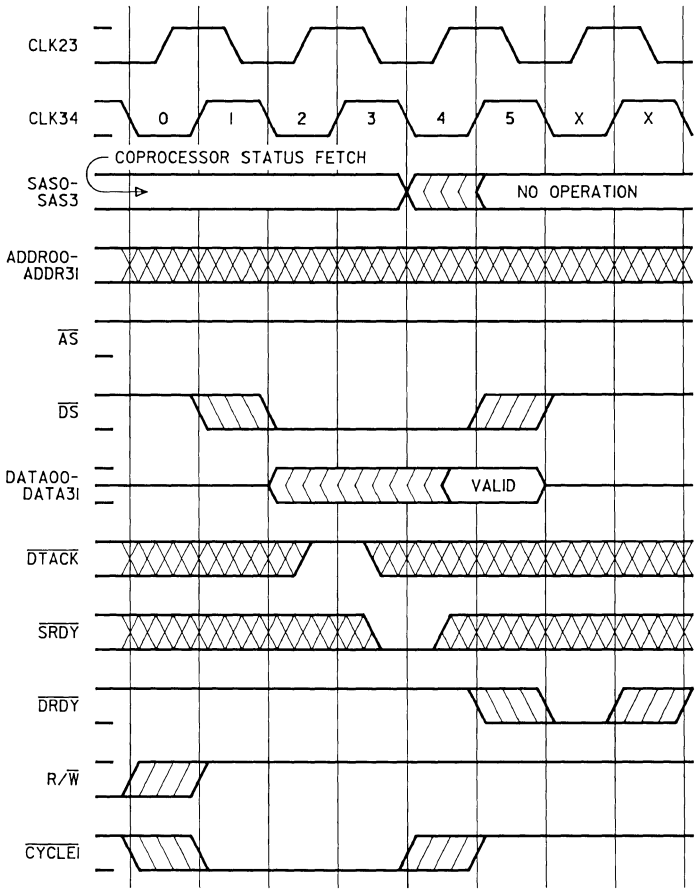


Note: Zero wait cycles use of $\overline{\text{DTACK}}$ or $\overline{\text{SRDY}}$

Figure 2-40. Coprocessor Operand Fetch

2.14.3 Coprocessor Status Fetch

After processing the data latched during the coprocessor operand fetch transaction, the coprocessor indicates that it is finished by asserting the coprocessor done input ($\overline{\text{DONE}}$) of the CPU. Approximately two clock cycles later, the CPU initiates the coprocessor status fetch transaction shown on Figure 2-41. This is a read type transaction where the coprocessor drives the data bus with status information. There is no $\overline{\text{AS}}$ issued to keep the memory from being accessed. The SAS code is "coprocessor status fetch."

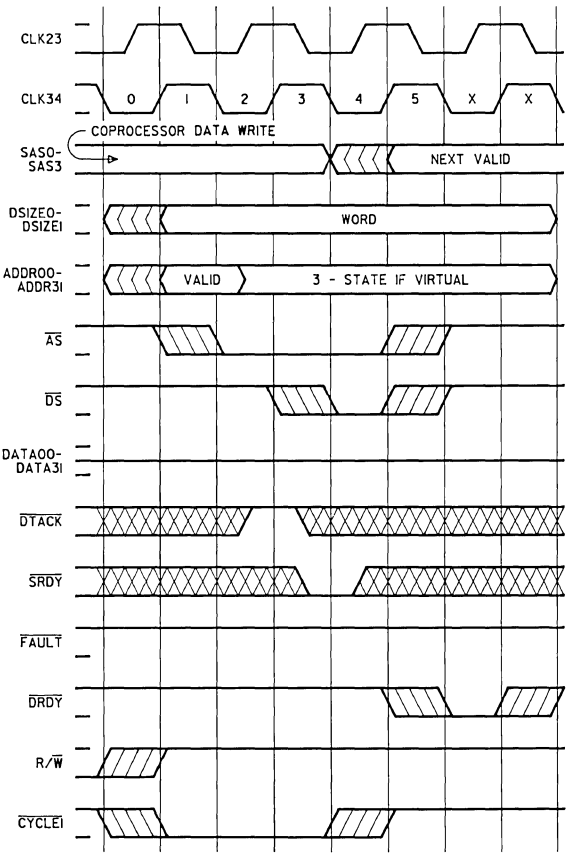


Note: Coprocessor status fetch begins approximately two clock cycles after the microprocessor's processor done input ($\overline{\text{DONE}}$) has been driven low.

Figure 2-41. Coprocessor Status Read (Using $\overline{\text{SRDY}}$)

2.14.4 Coprocessor Data Write

After doing the coprocessor status fetch, the CPU will perform from zero to three coprocessor data write transactions, depending on what coprocessor instruction is being executed. For this transaction the CPU goes through the motions of doing a write to the memory but the coprocessor drives the data bus with the results that it wants to send to the memory. The CPU does not drive the data bus during this transaction. The SAS is "coprocessor data write," and DSIZE is a word. The memory is issuing the acknowledge for this transaction. Figure 2-42 shows the protocol for a single coprocessor data write.



Notes:
1. Zero wait cycles using $\overline{\text{SRDY}}$.
2. DATA00—DATA31 supplied by coprocessor.

Figure 2-42. Coprocessor Data Write

2.15 EXCEPTIONAL CONDITIONS

In addition to interrupts and reset requests, several types of events may interrupt the execution of a program. The four events, called exceptional conditions, are: normal exceptions, stack exceptions, process exceptions, and reset exceptions. When an exception occurs, the microprocessor sets the 4-bit internal state code (ISC) field and the 2-bit exception type (ET) field in the processor status word to identify the exception. (Table 2-8 lists the exception conditions and their respective ISC codes.) The microprocessor also executes the appropriate microsequence before passing control to the operating system. These sequences save the context of the current process and give the operating system information it needs to locate the correct exception handler. The saved context enables the program to resume execution after the exception is handled.

Table 2-8. Exception Conditions

Exception Type	Exception	Internal State Code Bit			
		6	5	4	3
Normal Exception (ET=11)	Integer zero-divide	0	0	0	0
	Trace trap	0	0	0	1
	Illegal opcode	0	0	1	0
	Reserved opcode	0	0	1	1
	Invalid descriptor	0	1	0	0
	External memory fault	0	1	0	1
	Gate vector fault	0	1	1	0
	Illegal level change	0	1	1	1
	Reserved data type	1	0	0	0
	Integer overflow	1	0	0	1
	Privileged opcode	1	0	1	0
	Breakpoint trap	1	1	1	0
	Privileged register	1	1	1	1
Stack Exception (ET=01)	Stack bound	0	0	0	0
	Stack fault	0	0	0	1
	Interrupt ID fetch	0	0	1	1
Process Exception (ET=10)	Old PCB fault	0	0	0	0
	Gate PCB fault	0	0	0	1
	New PCB fault	0	1	0	0
Reset Exception (ET=00)	Old PCB fault	0	0	0	0
	System data	0	0	0	1
	Interrupt stack fault	0	0	1	0
	External reset	0	0	1	1
	New PCB fault	0	1	0	0
	Gate vector fault	0	1	1	0

* These exceptions reset the processor status word flags.

The exceptions increase in levels of severity, with normal exceptions being the least severe and reset exceptions being the most severe. An exception (but not reset exceptions which require restarting the system) can ripple up through levels of exception severity if its handling routine cannot resolve the condition that caused the exception.

1. **Normal Exception**. The microprocessor generates this class of exception when it detects a condition such as trap, invalid opcode, incorrect address mode, or illegal operation. Most normal exceptions occur during the translation or execution of an instruction.
2. **Stack Exception**. This exception may occur during a process switch or a GATE sequence (see Chapter 4).
3. **Process Exception**. This exception may occur during a process switch (see Chapter 4).
4. **Reset Exception**. This exception is triggered by an error condition in accessing critical system data and requires restarting of the system. Since exceptions can ripple up to higher levels of severity, reset exceptions may occur during reset and also during process and normal exceptions. The microprocessor reacts as if an external reset occurred when a reset exception is detected. (See **2.11 Reset** and **Chapter 4. OPERATING SYSTEM CONSIDERATIONS.)**

Normal exceptions consist of two types of events generated by the microprocessor - traps and exceptions. When a trap is generated, the instruction that caused the trap is executed completely, and the program counter (PC) points to the next executable instruction. (Integer overflows may not behave this way due to pipelining; see part b under **Integer Overflow**.) When an exception is generated, the PC points to the opcode of the instruction that caused the exception; this instruction may have been executed partially or not at all. Each different trap or exception uses a different trap vector to branch to the corresponding trap or exception-handling software.

There are three kinds of traps:

1. **Breakpoint Trap (BPT)**. This trap is invoked whenever the breakpoint trap (BPT) instruction is executed.
2. **Integer Overflow**. This trap is enabled when the enable overflow trap (OE) bit in the processor status word is set. Overflow trapping behaves as follows:
   a. When an overflow trap occurs, the OE bit is cleared before the PSW is saved.
   b. When an overflow trap occurs, the instruction following the instruction that caused the overflow trap may or may not be executed before the microsequence is entered. Consequently, the saved PC may point to the instruction following the trapped instruction or to the next instruction after that one. If the instruction following the trapped instruction is completed, it may not set the PSW flags.
   c. If two consecutive instructions cause overflow traps, only one overflow trap occurs.
   d. An overflow trap occurs if the OE bit is set and the execution of an instruction causes the V (overflow) bit in the PSW to be set (1) after the instruction is completed. In particular, this can be caused by the return from gate (RETG) and return to process (RETPS) instructions or by an explicit move to the PSW.

3. **Trace Trap**. Trace trapping is enabled when the trace enable (TE) bit in the PSW is set. This causes a trace trap to occur after each instruction is executed (except for the RETPS, CALLPS, and RETG instructions).

There are ten types of exceptions:

1. **External Memory Fault**. This exception occurs if alignment requirements are violated, if an external device asserts the $\overline{\text{FAULT}}$ input on an access, if a fault occurs during a coprocessor status fetch, or if no coprocessor responds to a support processor broadcast. Alignment fault behavior has the following properties:
   a. No alignment fault ever occurs on a byte access.
   b. No alignment fault ever occurs on an instruction fetch access.
   c. An alignment fault occurs if the access is a data access of word length and if address bit 1 (ADDR01) or address bit 0 (ADDR00) is 1.
   d. An alignment fault occurs if the access is a data access of halfword length and address bit 0 (ADDR00) is 1.
2. **Gate Vector Fault**. This exception is caused by a memory fault when reading gate tables during a gate (GATE) instruction.
3. **Illegal Level Change**. This exception is caused when attempting to increase the current execution privilege on a return from gate (RETG) instruction.
4. **Illegal Opcode**. The opcode is not defined for the microprocessor.
5. **Integer Zero-divide**. This exception is caused by an attempt to divide by zero and is always enabled. This exception resets the PSW flags.
6. **Invalid Descriptor**. The address mode generated cannot be used in the specified way. This exception resets the PSW flags and may result from the following causes:
   a. Literal or immediate used as destination.
   b. Effective address requested of literal or immediate.
   c. Effective address requested of a register.
7. **Privileged Opcode**. The opcode is defined for kernel execution level only. An attempt to execute it in another execution level causes this exception.
8. **Privileged Register**. An attempt to write the three privileged registers (process status word, process control block pointer, and interrupt stack pointer) in an execution level other than kernel causes this exception. This exception resets the PSW flags.
9. **Reserved Data Type**. The operand type described by the expanded operand-type descriptor is not implemented in the microprocessor. This exception resets the PSW flags.
10. **Reserved Opcode**. The opcode is not implemented on the microprocessor, but is reserved for future use.

2.16 TRACE MECHANISM

Every instruction for the *WE* 32100 Microprocessor consists of an interruptible and a noninterruptible portion. Because a trace trap is detected in the noninterruptible portion, trace traps have priority over interrupts. The microprocessor's trace trap mechanism uses two bits in the processor status word: trace enable (TE) and trace mask (TM).

Trace traps are enabled if TE is set (1), but a trace trap is generated only if TM is also set (see Table 2-9). In the table:

- TE-beg is the value of the TE bit at the start of an instruction.
- TE-end is the value of the TE bit at the time the trace trap is detected.
- TM-end is the value of the TM bit at the same instant the trace trap is detected.

The microprocessor detects a trace trap before the next instruction starts. Any of the following actions may change the values of the TE and TM bits at the end of an instruction:

- An instruction, other than an operating system instruction or microsequence, writes to the PSW and changes TE. However, this method of changing TE causes inconsistent trace behavior and should be avoided.
- A return from gate (RETG) instruction restores the PSW from the stack.
- A context switch to a process loads the PSW from the process control block.

Because of the way a return to process (RETPS), call process (CALLPS), or return from gate (RETG) instruction changes TE and TM when it overwrites the PSW, these instructions cannot be traced.

Table 2-9. Truth Table for Trace Trap

TE-beg	TE-end	TM-end	Trap
0	0	0 or 1	No
0	1	0	No
0	1	1	Yes
1	0	0 or 1	No
1	1	0	No
1	1	1	Yes

Note: This table is valid only if an operating system instruction or microsequence is used to alter the TE bit of the PSW.

The TM bit cannot be set by software. However, the microprocessor changes TM automatically by:

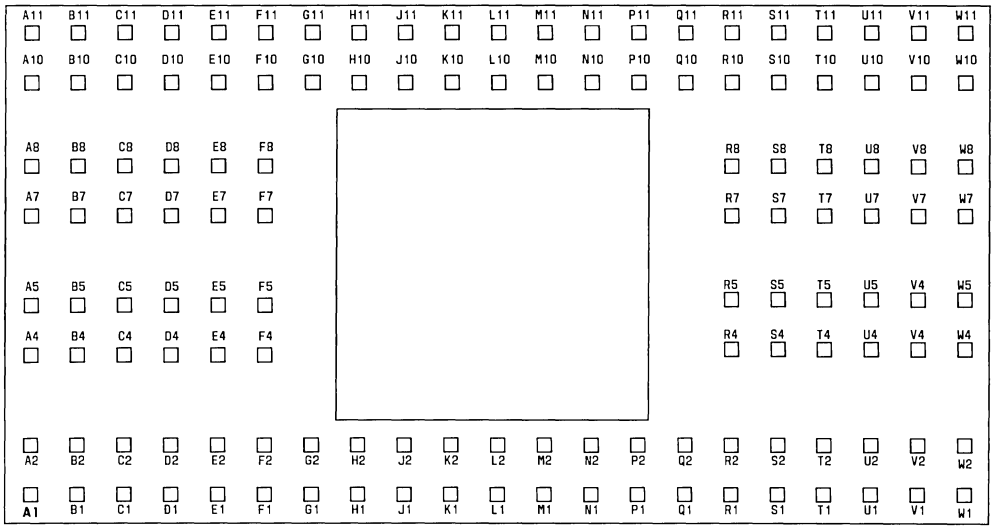
- Setting TM to 1 at the beginning of every instruction.
- Clearing TM to 0 as part of every microsequence that performs a context switch.
- Clearing TM to 0 as part of the return from gate microsequence.
- Clearing TM to 0 when it detects and responds to a fault or interrupt.

The TM bit masks the TE bit for the duration of one instruction. The user's trace-trap handler should use TM to prevent a trace trap when a return from gate instruction returns control to a process. Similarly, the microprocessor uses the TM bit to prevent a trace trap from occurring in the context of a newly switched process when the previous process is being traced.

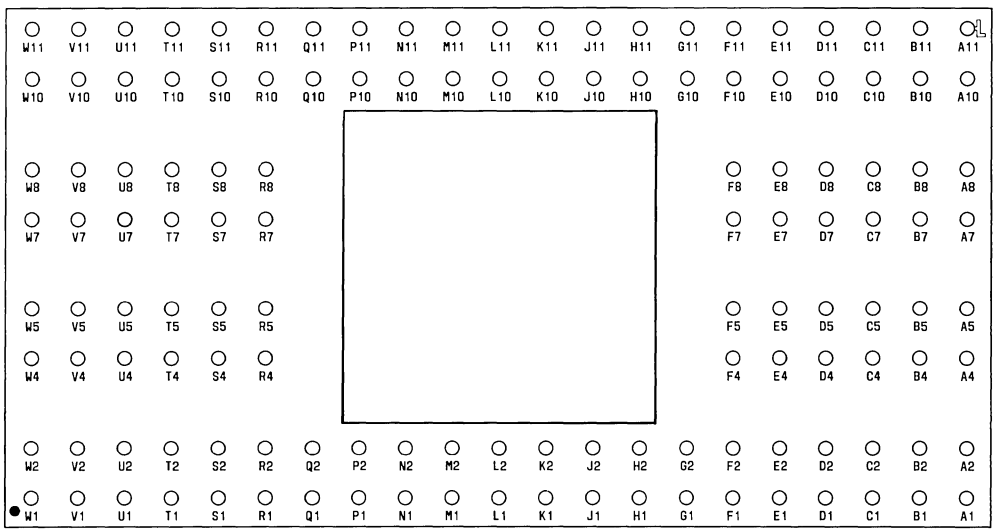
2.17 PIN ASSIGNMENTS

The *WE* 32100 Microprocessor contains 107 active pins, ten power pins, and eleven ground pins. Figure 2-43 illustrates the *WE* 32100 Microprocessor pin-array package as viewed from both the top and bottom. The top view shows the scratch pad test points and the 700 mil square heat sink attachment area. The scratch pads provide test points for each pin. The heat sink is user-supplied and is used in applications that require additional cooling. The following tables list the pins both in numerical order and by functional groups.

In the following pin function descriptions the term *asserted* means that a signal is driven to its active state either by the microprocessor (outputs) or an external device (inputs). The term *negated* means that the signal is driven to its inactive state. A bar over a signal name (e.g., $\overline{AS}$) indicates that the signal is active low, logic 0. The 0 bit is the least significant bit for signals which occupy two or more pins (e.g., DSIZE0—DSIZE1). The signal type column is interpreted as input (I), output (O), or bidirectional (I/O).



Top View



Bottom View

Figure 2-43. *WE* 32100 Microprocessor Pin Configuration

Table 2-10. *WE* 32100 Microprocessor Pin Descriptions			
Pin	**Name**	**Type**	**Description**
A1	DATA18	I/O	Microprocessor Data 18
A2	ADDR17	O	Microprocessor Address 17
A4	DATA17	I/O	Microprocessor Data 17
A5	DATA14	I/O	Microprocessor Data 14
A7	ADDR12	O	Microprocessor Address 12
A8	ADDR11	O	Microprocessor Address 11
A10	DATA08	I/O	Microprocessor Data 08
A11	ADDR06	O	Microprocessor Address 06
B1	DATA19	I/O	Microprocessor Data 19
B2	GRD	–	Microprocessor Ground
B4	DATA16	I/O	Microprocessor Data 16
B5	ADDR13	O	Microprocessor Address 13
B7	ADDR10	O	Microprocessor Address 10
B8	ADDR09	O	Microprocessor Address 09
B10	ADDR05	O	Microprocessor Address 05
B11	ADDR04	O	Microprocessor Address 04
C1	DATA22	I/O	Microprocessor Data 22
C2	DATA20	I/O	Microprocessor Data 20
C4	DATA15	I/O	Microprocessor Data 15
C5	+5V	–	Microprocessor Power
C7	DATA09	I/O	Microprocessor Data 09
C8	+5V	–	Microprocessor Power
C10	DATA04	I/O	Microprocessor Data 04
C11	DATA03	I/O	Microprocessor Data 03
D1	ADDR23	O	Microprocessor Address 23
D2	+5V	–	Microprocessor Power
D4	GRD	–	Microprocessor Ground
D5	DATA05	I/O	Microprocessor Data 05
D7	GRD	–	Microprocessor Ground
D8	ADDR07	O	Microprocessor Address 07
D10	GRD	–	Microprocessor Ground
D11	DATA02	I/O	Microprocessor Data 02
E1	DATA12	I/O	Microprocessor Data 12
E2	DATA11	I/O	Microprocessor Data 11
E4	ADDR08	O	Microprocessor Address 08
E5	DATA06	I/O	Microprocessor Data 06
E7	ADDR03	O	Microprocessor Address 03
E8	ADDR01	O	Microprocessor Address 01
E10	IPL1	I	Interrupt Priority Level 1
E11	DATA01	I/O	Microprocessor Data 01

Table 2-10. *WE* 32100 Microprocessor Pin Descriptions (Continued)			
Pin	**Name**	**Type**	**Description**
F1	ADDR14	O	Microprocessor Address 14
F2	DATA13	I/O	Microprocessor Data 13
F4	DATA10	I/O	Microprocessor Data 10
F5	DATA07	I/O	Microprocessor Data 07
F7	ADDR02	O	Microprocessor Address 02
F8	ADDR00	O	Microprocessor Address 00
F10	+5V	—	Microprocessor Power
F11	DATA00	I/O	Microprocessor Data 00
G1	ADDR15	O	Microprocessor Address 15
G2	GRD	—	Microprocessor Ground
G10	IPL3	I	Interrupt Priority Level 3
G11	$\overline{\text{VAD}}$	O	Virtual Address
H1	ADDR18	O	Microprocessor Address 18
H2	ADDR16	O	Microprocessor Address 16
H10	$\overline{\text{AVEC}}$	I	Auto-vector
H11	IPL0	I	Interrupt Priority Level 0
J1	ADDR19	O	Microprocessor Address 19
J2	+5V		Microprocessor Power
J10	IPL2	I	Interrupt Priority Level 2
J11	$\overline{\text{INTOPT}}$	I	Interrupt Option
K1	ADDR20	O	Microprocessor Address 20
K2	DATA21	I/O	Microprocessor Address 21
K10	$\overline{\text{NMINT}}$	I	Nonmaskable Interrupt
K11	—	—	**WARNING**: This pin is for manufacturing use only and must be tied high (+5 Vdc).
L1	ADDR21	O	Microprocessor Address 21
L2	ADDR22	O	Microprocessor Address 22
L10	$\overline{\text{ABORT}}$	O	Access Abort
L11	$\overline{\text{DRDY}}$	O	Data Ready
M1	DATA23	I/O	Microprocessor Data 23
M2	DATA25	I/O	Microprocessor Data 25
M10	CLK34	I	Input Clock 34
M11	$\overline{\text{AS}}$	O	Address Strobe
N1	DATA24	I/O	Microprocessor Data 24
N2	GRD	—	Microprocessor Ground
N10	CLK23	I	Input Clock 23
N11	$\overline{\text{DS}}$	O	Data Strobe
P1	DATA26	I/O	Microprocessor Data 26
P2	ADDR28	O	Microprocessor Address 28
P10	$\overline{\text{FAULT}}$	I	Fault
P11	$\overline{\text{RESETR}}$	I	Reset Request

Table 2-10. *WE* 32100 Microprocessor Pin Descriptions (Continued)			
Pin	**Name**	**Type**	**Description**
Q1	ADDR27	O	Microprocessor Address 27
Q2	+5V	–	Microprocessor Power
Q10	$\overline{\text{RESET}}$	O	Reset Acknowledge
Q11	$\overline{\text{BLKFTCH}}$	I	Block (Double Word) Fetch
R1	ADDR29	O	Microprocessor Address 29
R2	ADDR30	O	Microprocessor Address 30
R4	DATA31	I/O	Microprocessor Data 31
R5	IQS1	O	Instruction Queue Status 1
R7	$\overline{\text{SAS2}}$	O	Access Status Code 2
R8	$\overline{\text{SRDY}}$	I	Synchronous Ready
R10	$\overline{\text{RETRY}}$	I	Retry
R11	$\overline{\text{DTACK}}$	I	Data Transfer Acknowledge
S1	ADDR24	O	Microprocessor Address 24
S2	ADDR31	O	Microprocessor Address 31
S4	DATA30	I/O	Microprocessor Data 30
S5	XMD1	O	Execution Mode 1
S7	$\overline{\text{BRACK}}$	I/O	Bus Request Acknowledge
S8	DSIZE1	O	Data Size 1
S10	GRD	–	Microprocessor Ground
S11	$\overline{\text{STOP}}$	I	32100 Stop
T1	ADDR25	O	Microprocessor Address 25
T2	GRD	–	Microprocessor Ground
T4	+5V	–	Microprocessor Power
T5	XMD0	O	Execution Mode 0
T7	+5V	–	Microprocessor Power
T8	$\overline{\text{SAS0}}$	O	Access Status Code 0
T10	$\overline{\text{DSHAD}}$	I	Data Bus Shadow
T11	$\overline{\text{CYCLEI}}$	O	Cycle Initiate
U1	ADDR26	O	Microprocessor Address 26
U2	DATA27	I/O	Microprocessor Data 27
U4	IQS0	O	Instruction Queue Status 0
U5	GRD	–	Microprocessor Ground
U7	R/$\overline{\text{W}}$	O	Read/Write
U8	GRD	–	Microprocessor Ground
U10	+5V	–	Microprocessor Power
U11	No Connect	–	**WARNING:** This pin must be left unconnected.
V1	DATA28	I/O	Microprocessor Data 28
V2	+5V	–	Microprocessor Power
V4	$\overline{\text{SOI}}$	O	Start of Instruction
V5	$\overline{\text{BUSRQ}}$	I/O	Bus Request

Table 2-10. *WE* 32100 Microprocessor Pin Descriptions (Continued)			
Pin	**Name**	**Type**	**Description**
V7	DSIZE0	O	Data Size 0
V8	$\overline{\text{HIGHZ}}$	I	High Impedance
V10	$\overline{\text{RRREQ}}$	I	Relinquish and Retry Request
V11	$\overline{\text{BARB}}$	I	Bus Arbiter
W1	GRD	—	Microprocessor Ground
W2	DATA29	I/O	Microprocessor Data 29
W4	SAS3	O	Access Status Code 3
W5	SAS1	O	Access Status Code 1
W7	$\overline{\text{DONE}}$	I	Coprocessor Done
W8	$\overline{\text{RRRACK}}$	O	Relinquish and Retry Request Acknowledge
W10, W11	No Connect	—	**WARNING:** These pins must be left unconnected.

Table 2-11. Address and Data Signals			
Name	**Pin(s)**	**Type**	**Description**
ADDR00—ADDR31	F8,E8,F7,E7, B11,B10,A11, D8,E4,B8,B7, A8,A7,B5,F1, G1,H2,A2,H1, J1,K1,L1,L2, D1,S1,T1,U1, Q1,P2,R1,R2, S2	O	**Address.** These pins are used by the microprocessor to issue 32-bit addresses for off-chip accesses. They also convey the interrupt acknowledge level on bits 2 through 6 during an interrupt acknowledge operation.
DATA00—DATA31	F11,E11,D11, C11,C10,D5, E5,F5,A10, C7,F4,E2,E1, F2,A5,C4,B4, A4,A1,B1,C2, K2,C1,M1,N1, M2,P1,U2,V1, W2,S4,R4	I/O	**Data.** These bidirectional pins are used to convey data to and from the microprocessor. This data may be an interrupt vector (bits 0 through 7).

Table 2-12. Interface and Control Signals

Name	Pin(s)	Type	Description
$\overline{AS}$	M11	O	**Address Strobe.** When low (0), this signal indicates the presence of a valid physical address on the address pins. If the address is virtual, the falling edge of $\overline{AS}$ indicates a valid address, and the address pins are 3-stated subsequent to the falling edge of $\overline{AS}$.
$\overline{CYCLEI}$	T11	O	**Cycle Initiate.** This signal is asserted at the beginning of a bus transaction and negated two clock cycles later. $\overline{CYCLEI}$ is asserted in both the read and write halves of an interlocked read transaction.
$\overline{DONE}$	W7	I	**Coprocessor Done.** This input is recognized during a coprocessor instruction. It informs the microprocessor that a slave processor has completed its operation.
$\overline{DRDY}$	L11	O	**Data Ready.** When asserted, this signal indicates that the microprocessor has not detected any bus exceptions ($\overline{FAULT}$, $\overline{RETRY}$, $\overline{RRREQ}$ signals) during the current bus cycle. The trailing edge of this signal indicates the end of a bus transaction which has no bus exceptions.
$\overline{DS}$	N11	O	**Data Strobe.** During a read operation this signal, when low, indicates that a slave device can place data on the data bus. During a write operation, this signal, when low, indicates that the microprocessor has placed valid data on the data bus.
$\overline{DTACK}$	R11	I	**Data Transfer Acknowledge.** This signal is used to handshake between the microprocessor and a slave device. During a read operation, the microprocessor latches data present on the data bus and terminates the bus transaction one cycle after $\overline{DTACK}$ is driven low by a slave device. During a write operation, the transaction is terminated when a slave device drives $\overline{DTACK}$ low. If $\overline{DTACK}$ is high, wait states are inserted in current cycle. $\overline{DTACK}$ is ignored if the data bus shadow ($\overline{DSHAD}$) input is asserted. The $\overline{DTACK}$ input can be returned asynchronously and is double latched to avoid metastability.
$\overline{SRDY}$	R8	I	**Synchronous Ready.** When asserted, this signal is a synchronous input that begins the termination of a read or write operation. It is sampled only once on the leading edge of the fifth clock state during read and write operations. If $\overline{SRDY}$ is not asserted at this time and $\overline{DTACK}$ was not asserted during the previous cycle, then wait-state cycles are inserted until either signal is asserted. $\overline{SRDY}$ is ignored if the data bus shadow ($\overline{DSHAD}$) input was previously asserted.

Table 2-13. Access Status Signals

Name	Pin(s)	Type	Description
$\overline{\text{BLKFTCH}}$	Q11	I	**Block (Double-Word) Fetch.** This input indicates to the microprocessor that the memory system can perform a double-word (eight byte) program block fetch. On all instruction fetches, the data size (DSIZE0 and DSIZE1) pins will show a double-word access. If the memory system can handle a double-word access, it can activate this input. Otherwise, the input is left inactive, and the microprocessor fetches a block of instructions by two consecutive reads.
DSIZE0—DSIZE1	V7,S8	O	**Data Size.** This two-bit output is used to indicate whether the microprocessor is transferring byte, halfword, word, or double-word data in the current bus transaction. On all instruction fetches, the DSIZE0—DSIZE1 pins will have the value for double word. **DSIZE1 DSIZE0 Description** 0 0 Word transaction 0 1 Double-word transaction 1 0 Halfword transaction 1 1 Byte transaction
R/$\overline{\text{W}}$	U7	O	**Read/Write.** This signal indicates whether the bus transaction is a read or a write. When low (0), the operation is a write. When high (1), the operation is a read. This pin is valid during the time the address strobe ($\overline{\text{AS}}$) is active.
SAS0—SAS3	T8,W5 R7,W4	O	**Access Status Codes.** These pins describe the type of bus transaction being executed. SAS0 is the least significant bit of the access status codes. **SAS3 SAS2 SAS1 SAS0 Description** 0 0 0 0 Move translated word 0 0 0 1 Coprocessor data write 0 0 1 0 Auto-vector interrupt acknowledge 0 0 1 1 Coprocessor data fetch 0 1 0 0 Stop acknowledge 0 1 0 1 Coprocessor broadcast 0 1 1 0 Coprocessor status fetch 0 1 1 1 Read interlocked 1 0 0 0 Address fetch 1 0 0 1 Operand fetch 1 0 1 0 Write 1 0 1 1 Interrupt acknowledge 1 1 0 0 Instruction fetch after PC discontinuity 1 1 0 1 Instruction prefetch 1 1 1 0 Instruction fetch 1 1 1 1 No operation

Table 2-13. Access Status Signals (Continued)

Name	Pin(s)	Type	Description
$\overline{\text{VAD}}$	G11	O	**Virtual Address.** When low, this signal indicates that the address is virtual. When $\overline{\text{VAD}}$ is high, the address is a physical address. $\overline{\text{VAD}}$ is a level signal. It is asserted by execution of the enable virtual pin and jump (ENBVJMP) instruction and negated by execution of the disable virtual pin and jump (DISVJMP) instruction. Note: This output is the only indication of the CPU's mode of operation.
XMD0–XMD1	T5,S5	O	**Execution Mode.** These two outputs indicate the present execution mode of the microprocessor. **XMD1** **XMD0** **Description** 0 0 Kernel mode 0 1 Executive mode 1 0 Supervisor mode 1 1 User mode If a memory management unit (MMU) is present in the system, it may latch and use a spurious execution mode value if XMD0–XMD1 changes during an access. Since XMD0–XMD1 reflect the state of the current execution level (CM) bits in the PSW, changes to the CM field via non-microsequence instructions must be avoided. See Chapter 4 for a more detailed explanation of the execution modes.

Table 2-14. Interrupt Signals			
Name	**Pin(s)**	**Type**	**Description**
$\overline{\text{AVEC}}$	H10	I	**Auto-vector**. When this input is asserted with the interrupt priority level input, the microprocessor supplies its own vector. The vector number value is the inverted interrupt option input (INTOPT) concatenated with the interrupt priority level value. When auto-vector is not asserted, the interrupting device supplies the vector (see **2.9 Interrupts**).
$\overline{\text{INTOPT}}$	J11	I	**Interrupt Option**. This asynchronous input is latched along with interrupt priority level inputs IPL0—IPL3. It is then inverted and output on ADDR06 during an interrupt acknowledge transaction.
IPL0—IPL3	H11,E10, J10,G10	I	**Interrupt Priority Level**. These asynchronous inputs indicate the level of the pending interrupt. The code is based on a decreasing priority scheme with 0000 having the highest priority and 1110 the lowest. Level 1111 indicates no interrupts pending. To be acknowledged, the inversion of the requesting level on the pins must be greater than the present interrupt priority level (IPL field) in the process status word. The exception to this is a nonmaskable interrupt which can interrupt the microprocessor regardless of the present IPL field priority level. IPL0 is the LSB of the interrupt priority level code. (See **2.9 Interrupts.**)
$\overline{\text{NMINT}}$	K10	I	**Nonmaskable Interrupt**. When asserted, this asynchronous input indicates that a nonmaskable interrupt is being requested. The microprocessor acknowledges this interrupt with an auto-vector interrupt acknowledge cycle (see **2.9 Interrupts**). During the acknowledge cycle the microprocessor address bus contains all zeros.

Table 2-15. Arbitration Signals

Name	Pin(s)	Type	Description
$\overline{\text{BARB}}$	V11	I	**Bus Arbiter**. When this input is strapped low, the microprocessor is the arbiter of the bus. As arbiter, the microprocessor need not request access to the bus. When the pin is strapped high, the microprocessor is not the arbiter and must request bus access to use the bus. When the microprocessor is not the bus arbiter the following outputs are 3-stated until the CPU does a bus transaction: $\overline{\text{ABORT}}$ DATA00–DATA31 $\overline{\text{SAS0}}$–SAS3 ADDR00– $\overline{\text{DRDY}}$ $\overline{\text{VAD}}$ ADDR31 $\overline{\text{DS}}$ XMD0–XMD1 $\overline{\text{AS}}$ DSIZE0–DSIZE1 $\overline{\text{CYCLEI}}$ R/$\overline{\text{W}}$
$\overline{\text{BRACK}}$	S7	I/O	**Bus Request Acknowledge**. This signal is an output if the microprocessor is the arbiter of the bus and an input if it is not. As an output, this pin indicates that the bus request ($\overline{\text{BUSRQ}}$) has been recognized, and the microprocessor has 3-stated the bus for the requesting bus master. The bus signals which are 3-stated when the $\overline{\text{BRACK}}$ is issued are: $\overline{\text{ABORT}}$ DATA00–DATA31 $\overline{\text{SAS0}}$–SAS3 ADDR00– $\overline{\text{DRDY}}$ $\overline{\text{VAD}}$ ADDR31 $\overline{\text{DS}}$ XMD0–XMD1 $\overline{\text{AS}}$ DSIZE0–DSIZE1 $\overline{\text{CYCLEI}}$ R/$\overline{\text{W}}$ As an input, this pin indicates that the microprocessor's bus request has been recognized and the microprocessor may take possession of the bus.
$\overline{\text{BUSRQ}}$	V5	I/O	**Bus Request**. This asynchronous signal is an input if the microprocessor is the arbiter of the bus and an output if it is not. As an input, this signal indicates that an external device is requesting the bus. As an output, the signal indicates that the microprocessor is requesting the bus.

Table 2-16. Bus Exception Signals			
Name	**Pin(s)**	**Type**	**Description**
$\overline{\text{ABORT}}$	L10	O	**Access Abort.** This pin is asserted on an access that is to be ignored by the memory system. This occurs when the microprocessor has a program counter discontinuity with an instruction cache hit or an alignment fault.
$\overline{\text{DSHAD}}$	T10	I	**Data Bus Shadow.** This input is used by the memory management unit (MMU) to remove the microprocessor from the data bus. The DATA00—DATA31, $\overline{\text{DRDY}}$, DSIZE0—DSIZE1, and R/$\overline{\text{W}}$ pins are 3-stated when this input is asserted. When $\overline{\text{DSHAD}}$ is asserted, the $\overline{\text{DTACK}}$, $\overline{\text{SRDY}}$, and $\overline{\text{FAULT}}$ inputs are ignored.
$\overline{\text{FAULT}}$	P10	I	**Fault.** This input notifies the microprocessor that a fault condition has occurred. It is a double-latched, asynchronous input prior to the assertion of $\overline{\text{DTACK}}$, and synchronous after the assertion of $\overline{\text{DTACK}}$ (latched once). $\overline{\text{FAULT}}$ is ignored if $\overline{\text{DSHAD}}$ is asserted.
$\overline{\text{RESET}}$	Q10	O	**Reset Acknowledge.** This signal indicates that the microprocessor has recognized an external reset request, or that it has generated an internal reset (e.g., reset exception). The microprocessor executes its reset routine once it negates $\overline{\text{RESET}}$ (see **2.11 Reset**).
$\overline{\text{RESETR}}$	P11	I	**Reset Request.** This asynchronous signal is used to reset the microprocessor. $\overline{\text{RESETR}}$ must be active for three clock cycles in order to be acknowledged. The microprocessor acknowledges the request by immediately asserting $\overline{\text{RESET}}$.
$\overline{\text{RETRY}}$	R10	I	**Retry.** When this signal is asserted, the microprocessor terminates the current bus transaction and retries it when $\overline{\text{RETRY}}$ is negated.
$\overline{\text{RRRACK}}$	W8	O	**Relinquish and Retry Request Acknowledge.** This output is asserted in response to a relinquish and retry bus exception when the microprocessor has relinquished (3-stated) the bus. It is negated when the bus transaction terminated by the relinquish and retry bus exception is retried.

Table 2-16. Bus Exception Signals (Continued)

Name	Pin(s)	Type	Description
$\overline{\text{RRREQ}}$	V10	I	**Relinquish and Retry Request.** This signal is used to preempt a bus transaction so that the microprocessor bus may be used. The signal causes the microprocessor to terminate the current bus transaction and 3-state the following pins: $\overline{\text{ABORT}}$ DATA00–DATA31 $\overline{\text{SAS0}}$–$\overline{\text{SAS3}}$ ADDR00–ADDR31 $\overline{\text{DRDY}}$ $\overline{\text{VAD}}$ $\overline{\text{AS}}$ $\overline{\text{DS}}$ XMD0–XMD1 $\overline{\text{CYCLE1}}$ DSIZE0–DSIZE1 R/$\overline{\text{W}}$ The $\overline{\text{RRRACK}}$ signal is asserted after all the above listed pins are 3-stated. During this 3-state phase, the bus master requesting the relinquish and retry may take possession of the bus. No external bus arbitration signals are acknowledged during the assertion of a relinquish and retry request. When $\overline{\text{RRREQ}}$ is negated, the preempted bus transaction is retried.
$\overline{\text{STOP}}$	S11	I	**Stop.** When asserted, this asynchronous signal halts the execution of any further instructions beyond those already started. Before the microprocessor comes to a halt, there may be one more instruction beyond the instruction during which $\overline{\text{STOP}}$ was asserted.

Table 2-17. Development System Support Signals			
Name	**Pin(s)**	**Type**	**Description**
$\overline{\text{HIGHZ}}$	V8	I	**High Impedance.** When asserted, this signal puts all output pins on the microprocessor into the high-impedance state. This pin is intended for testing purposes.
IQS0—IQS1	U4,R5	O	**Instruction Queue Status.** This two-bit code indicates the activity on the microprocessor instruction queue. IQS0 is the least significant bit of the instruction queue status code. **Value** **Description** 00 Discard 4 bytes 01 Discard 1 byte 10 Discard 2 bytes 11 No discard this cycle
$\overline{\text{SOI}}$	V4	O	**Start of Instruction.** When asserted, this signal indicates that the microprocessor's internal control has fetched the opcode for the next instruction from the internal instruction queue. Since the instructions are pipelined, it does not always mean the end of the previous instruction execution.

Table 2-18. Clock Signals			
Name	**Pin(s)**	**Type**	**Description**
CLK34	M10	I	**Input Clock 34.** The falling edge of this clock signifies the beginning of a machine cycle. This clock input has the same frequency as CLK23 and lags it by 90°.
CLK23	N10	I	**Input Clock 23.** This clock input has the same frequency as CLK34 and leads it by 90°.

2.18 MICROPROCESSOR OPERATING REQUIREMENTS

The *WE* 32100 Microprocessor operates at a frequency of 10 MHz and requires only a single +5 volt supply. The operating requirements are summarized in Table 2-18. The following describes the microprocessor's electrical (inputs and outputs), clocking, and thermal requirements.

Note: Voltage level specifications are referenced to as either VCC (power supply input to the microprocessor) or GRD (microprocessor ground).

2.18.1 Electrical Requirements

The *WE* 32100 Microprocessor provides four classes of outputs: Classes 1, 2, 3, and 4. All classes can support TTL input voltage levels and are capable of driving loads of 130 pF.

Class 1: These outputs are capable of driving one TTL load or eight PNP Schottky TTL loads and have current allowance for an external holding resistor employed in 3-state buffers. The minimum holding resistor value is 2.7 kilohms. The Class 1 outputs are:

$\overline{\text{ABORT}}$	Access abort
$\overline{\text{AS}}$	Address strobe
$\overline{\text{BRACK}}$	Bus request acknowledge
$\overline{\text{BUSRQ}}$	Bus request
$\overline{\text{CYCLEI}}$	Cycle initiate
$\overline{\text{DRDY}}$	Data ready
$\overline{\text{DS}}$	Data strobe
DSIZE0—DSIZE1	Data size
IQS0—IQS1	Instruction queue status
R/$\overline{\text{W}}$	Read/write
$\overline{\text{RESET}}$	Reset acknowledge
SAS0—SAS3	Access status codes
$\overline{\text{SOI}}$	Start of instruction
$\overline{\text{VAD}}$	Virtual address
XMD0—XMD1	Execution mode

Class 2: This class has the same driving capabilities as Class 1, but does not have the current allowance for a holding resistor. The Class 2 outputs are:

ADDR00—ADDR31	Address bus
DATA00—DATA31	Data bus

Class 3: The signal in this class is an open drain output used for wired-logic operations, allowing more than one device to drive a node without conflict. An external resistor is required to pull this signal high. The minimum pull-up resistor value is 510 ohms. The Class 3 output is:

$\overline{\text{RRRACK}}$	Relinquish and retry request acknowledge

Class 4: This class is the same as Class 1; however, its minimum holding resistor value is 1.8 kilohms.

SAS0—SAS3	Access status codes

Table 2-19 contains the electrical specifications for the four classes of outputs.

The microprocessor has two types of inputs. The two clock inputs are CMOS inputs with CMOS voltage levels. The remaining inputs are CMOS with TTL voltage levels. The electrical specifications for both input types are given in Table 2-19.

2.18.2 Clocking Requirements

The microprocessor requires two input clocks (CLK34 and CLK23), both operating at a maximum frequency of 10 MHz. This frequency should not vary by more than ±0.02% for all temperature and power supply conditions. CLK34 lags CLK23 by 90 degrees, and its falling edge indicates the beginning of a machine cycle. The *WE* 32102 Clock is specifically designed for the CPU. The electrical specifications for the two clocks are given in Table 2-20.

2.18.3 Thermal Requirements

The ambient temperature at the microprocessor pins must be in the range of 0 °C to 70 °C. The microprocessor package provides a 700 mil square metalized pad for attachment of a heat sink for applications which require additional cooling. The heat sink must be supplied and attached by the user.

Table 2-19. Operating Requirements

Parameter		Symbol	Min	Nom	Max	Unit
Supply Voltage		V_{CC}	4.75	5.00	5.25	Vdc
Input Load Capacitance	TTL Inputs	C_{IN}	—	—	12	pF
	CMOS Clocks		—	—	7	pF
Total Output Load Capacitance	Class 1	C_L	—	—	130	pF
	Class 2		—	—	130	pF
	Class 3		—	—	130	pF
	Class 4		—	—	130	pF
Ambient Temperature at the Microprocessor Pins		T_A	0	—	70	°C
Humidity Range		—	5%	—	95%	—
Power Dissipation		P	—	—	0.8	W
Operating Frequency		F		—	10	MHz

Table 2-20. Output Electrical Specifications					
Outputs		**Min**	**Nom**	**Max**	**Units**
Output Sink Current (I_{OL}): ($V_{OL} \leq 0.4$ V)	Class 1	–	–	5.5	mAdc
	Class 2	–	–	3.5	mAdc
	Class 3*	–	–	10.0	mAdc
	Class 4	–	–	6.5	mAdc
Output Source Current (I_{OH}): ($V_{OH} \geq 2.4$ V)	Class 1	–	–	−5.5	mAdc
	Class 2	–	–	−3.5	mAdc
	Class 3*	–	–	−10.0	uAdc
	Class 4	–	–	−5.5	mAdc
Output Logic Levels	High Level	2.4	–	–	Vdc
	Low Level	–	–	0.4	Vdc

* See explanation of Class 3 in **2.15.1 Electrical Requirements.**

Table 2-21. Input Electrical Specifications					
Inputs		**Min**	**Nom**	**Max**	**Units**
TTL Input Voltage	High Level	2.0	–	V_{CC} + 0.5	Vdc
	Low Level	−0.5	–	0.8	Vdc
CMOS Clock Input Voltage	High Level	V_{CC} − 1.3	–	V_{CC} + 0.5	Vdc
	Low Level	0	–	0.8	Vdc
TTL Input Loading Current: (2.0 V $\leq V_{IH} \leq V_{CC}$)	High Level	0	–	0.01	mAdc
TTL Input Loading Current: (0 V $\leq V_{IL} \leq$ 0.8 V)	Low Level	−0.01	–	0	mAdc
CMOS Clock Input Loading Current: (V_{CC}−1.3 V $\leq V_{IH} \leq V_{CC}$)	High Level	0	–	0.01	mAdc
CMOS Clocks Input Loading Current: (0 V $\leq V_{IL} \leq$ 0.8 V)	Low Level	−0.01	–	0	mAdc

2.19 SUPPLEMENTARY PROTOCOL DIAGRAMS

The following supplementary protocol diagrams are provided:

Figure 2-44. Read Transaction Followed by a Read Transaction.

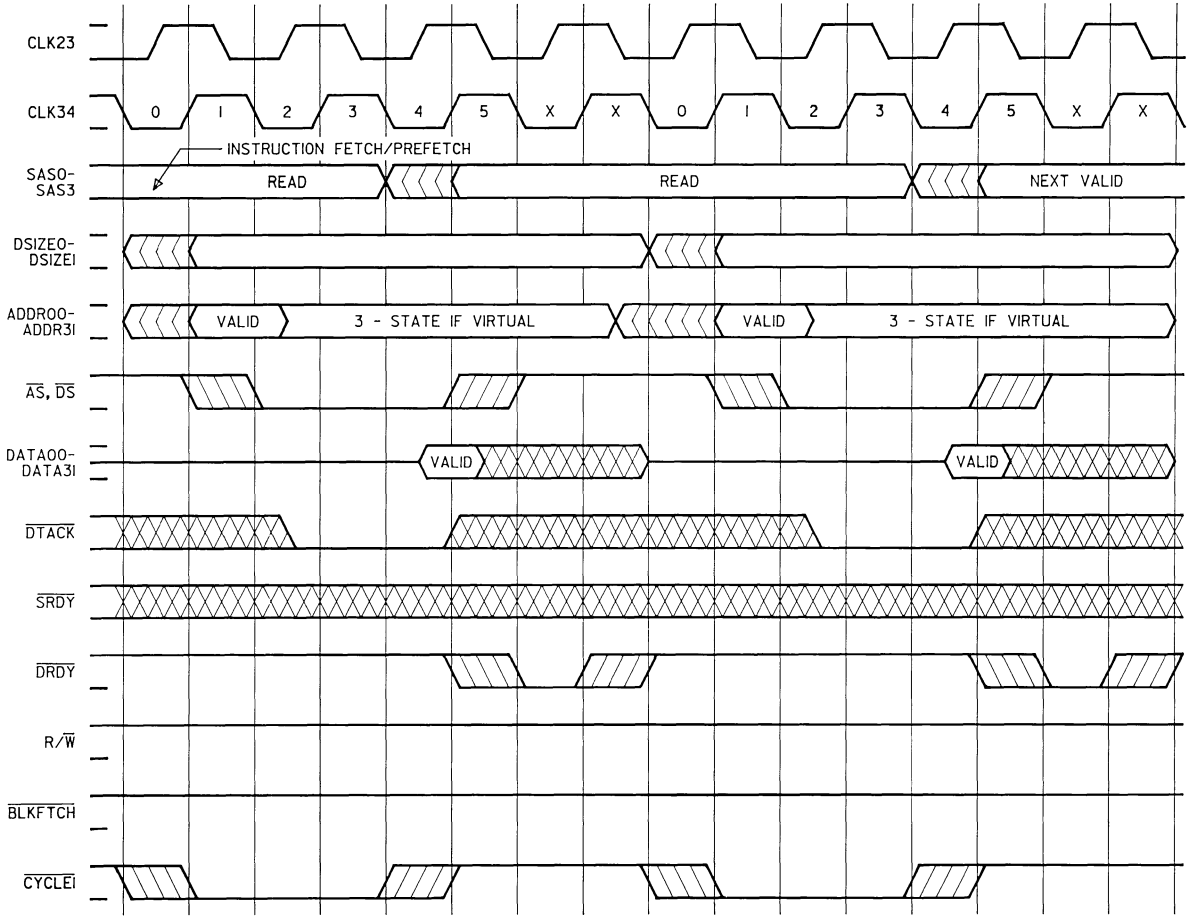
Figure 2-45. Read Transaction Followed by a Write Transaction (Using $\overline{\text{DTACK}}$).

Figure 2-46. Write Transaction Followed by a Write Transaction.

Figure 2-47. Write Transaction Followed by a Read Transaction.

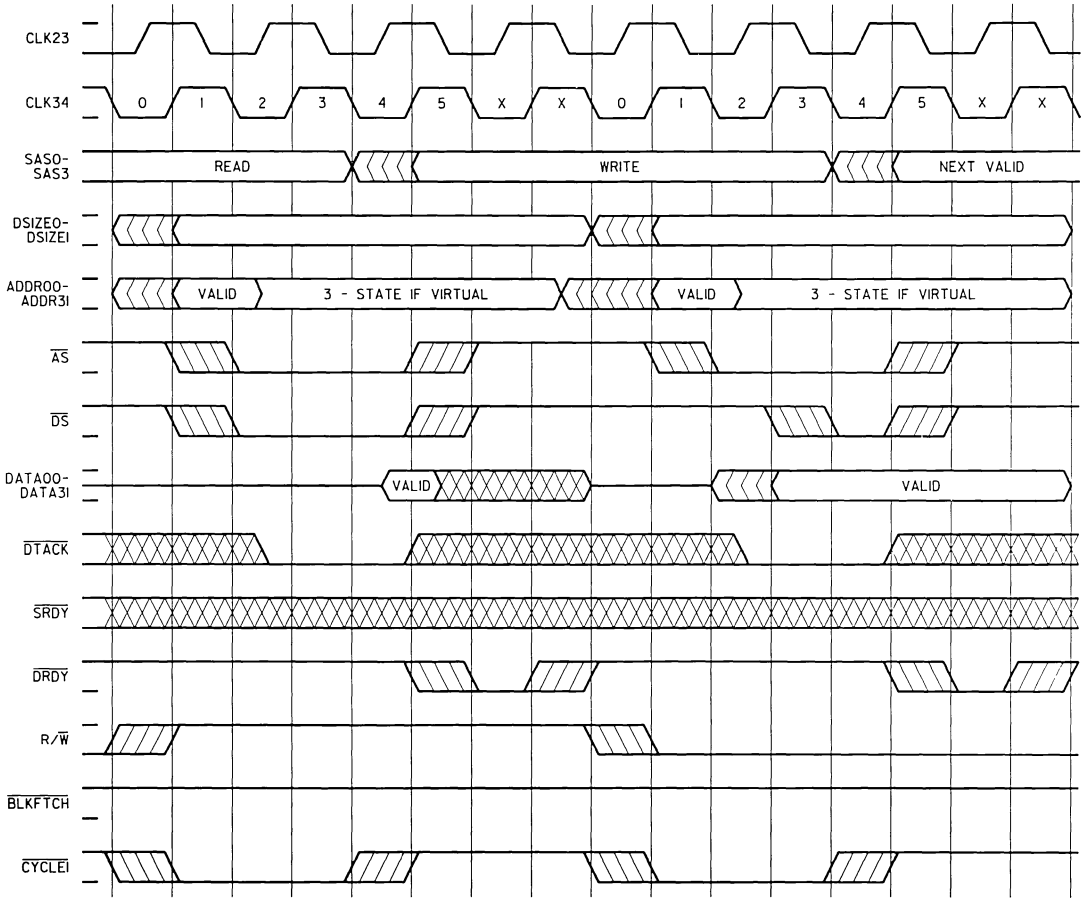
Figure 2-48. Double-Word Program Fetch Without Blockfetch Transaction (Using $\overline{\text{DTACK}}$).

Figure 2-49. Bus Arbitration During Relinquish and Retry.



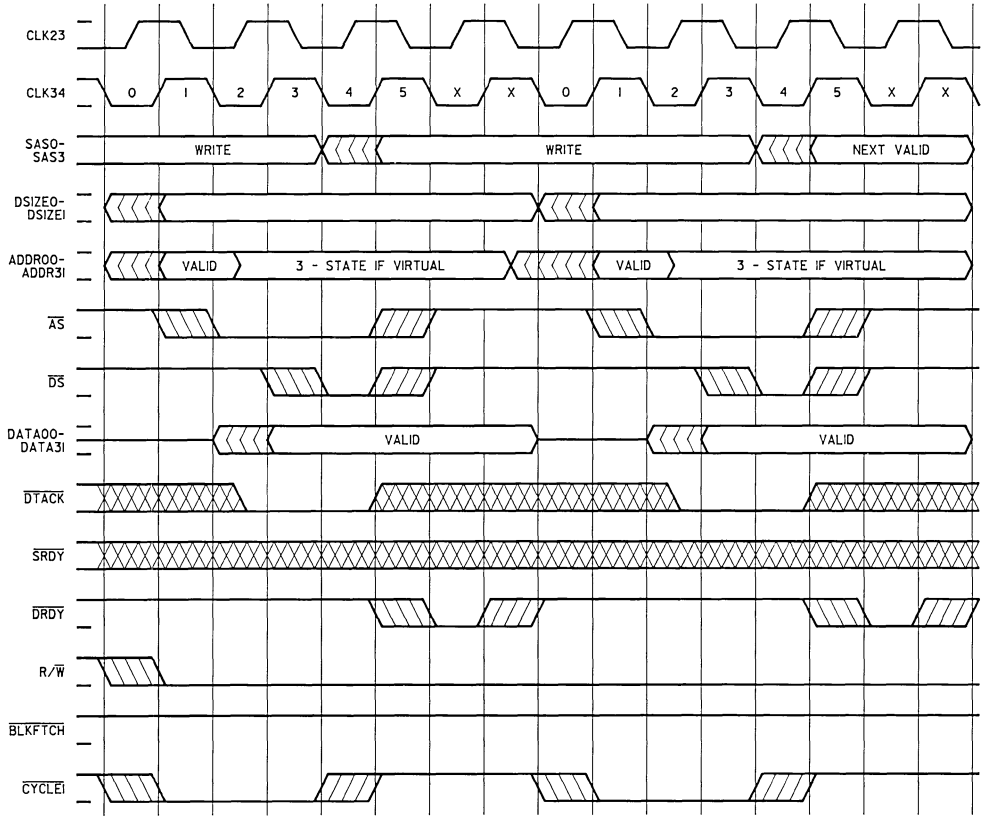
Note: Zero wait cycles.

Figure 2-44. Read Transaction Followed by a Read Transaction



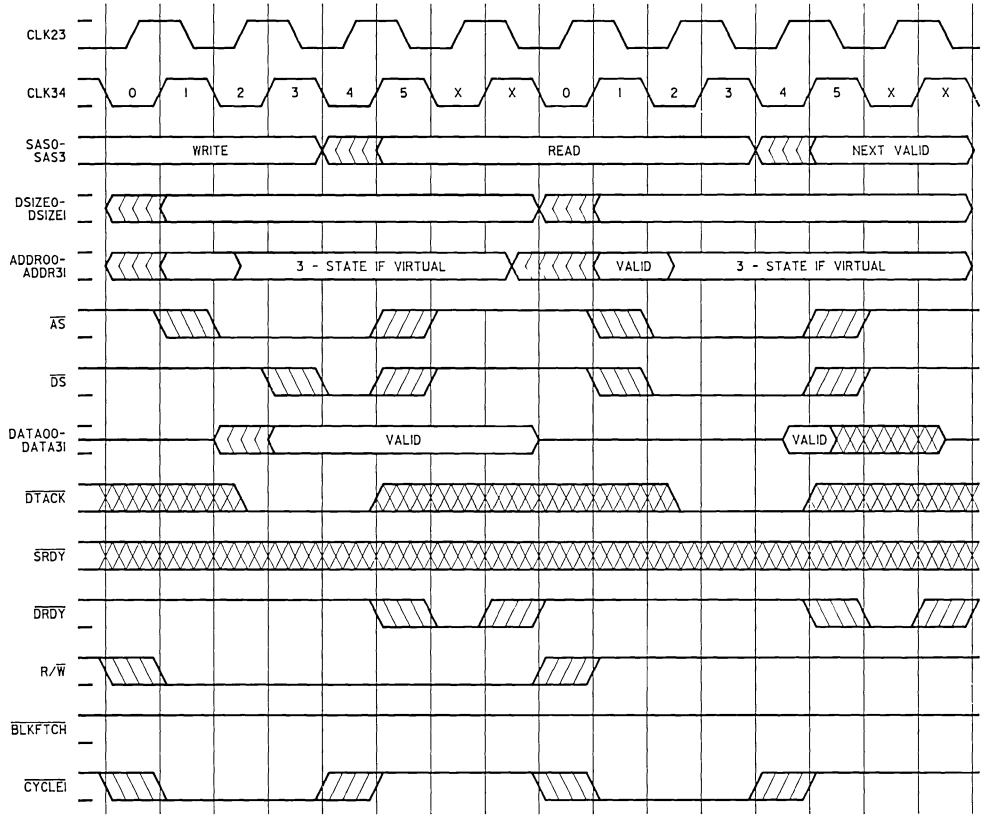
Note: Zero wait cycles.

Figure 2-45. Read Transaction Followed by a Write Transaction (Using DTACK)



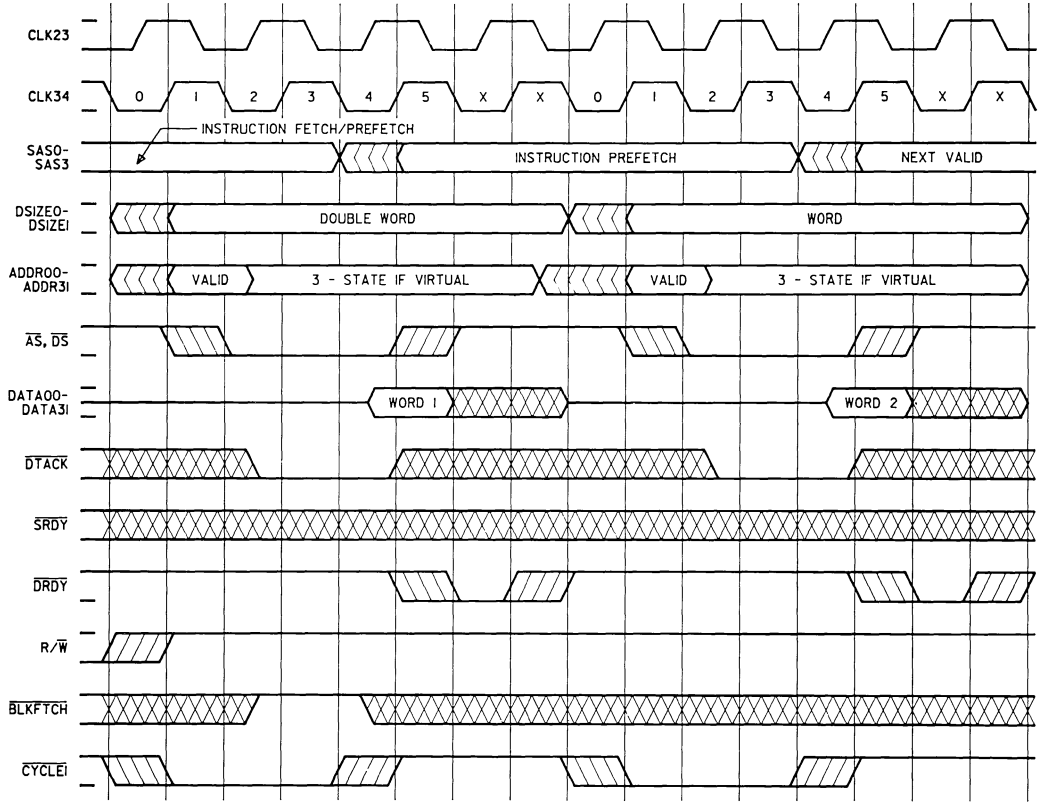
Note: Zero wait cycles.

Figure 2-46. Write Transaction Followed by a Write Transaction



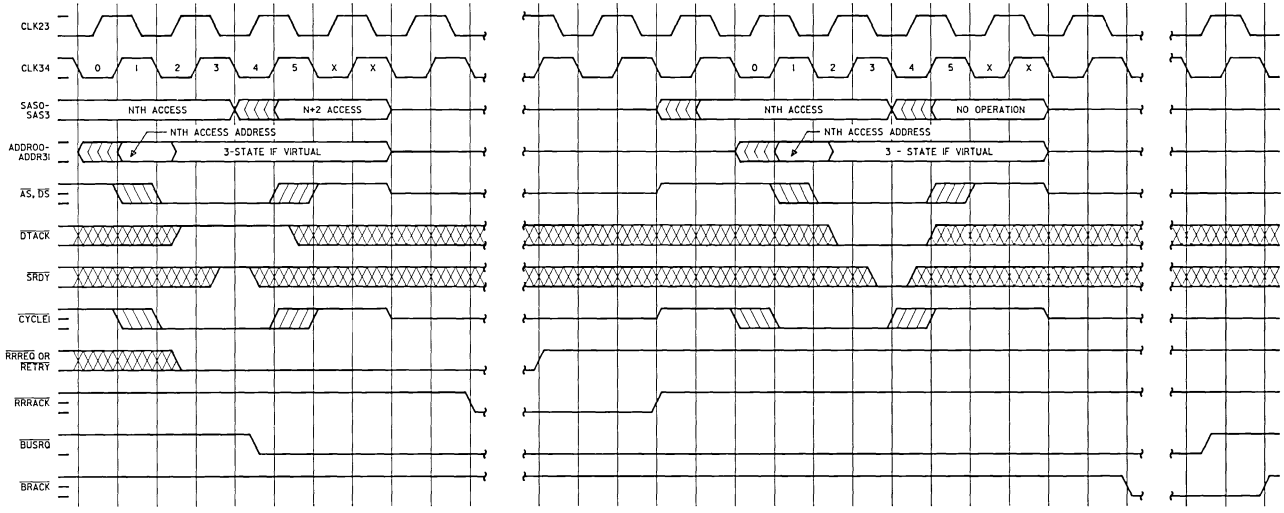
Note: Zero wait cycles.

Figure 2-47. Write Transaction Followed by a Read Transaction



Note: Zero wait cycles.

Figure 2-48. Double-Word Program Fetch Without Blockfetch Transaction (Using $\overline{\text{DTACK}}$)



Note: The same protocol diagram applies for retry and bus arbitration except that the address bus, data bus, and control signals are not 3-stated during the time $\overline{\text{RETRY}}$ is active and $\overline{\text{RRRACK}}$ is not issued.

Figure 2-49. Bus Arbitration During Relinquish and Retry

Chapter 3

Instruction Set and Addressing Modes

CHAPTER 3. INSTRUCTION SET AND ADDRESSING MODES

CONTENTS

CONTENTS

CONTENTS

3. INSTRUCTION SET

The *WE* 32100 Microprocessor has a powerful instruction set that includes the standard data transfer, arithmetic, and logical operations for microprocessors, plus some unique operating system operations. Its many program control instructions (branch, jump, return) provide flexibility for altering the sequence in which instructions are executed. Some of these instructions check the setting of the processor's condition flags before execution. For operating systems, the processor has instructions to establish an environment that permits other processes to take control of the processor. The special instructions dedicated to operating system use are discussed in Chapter 4.

The microprocessor instructions are mnemonic-based assembly language statements. However, programs may be written in C language and translated into assembly language by its C compiler.

A mnemonic defines the operation an instruction performs. For most arithmetic or logical operations, the mnemonic also defines one of the data types:

- **byte** - 8-bit data
- **halfword** - 16-bit data
- **word** - 32-bit data.

Some instructions perform operations on a *bit field*, a sequence of 1 to 32 bits contained in a word, or on a *block* (or *string*) of data locations.

3.1 DATA TYPES

The data types supported by the *WE* 32100 Microprocessor instruction set are illustrated on Figure 3-1 and are defined as:

byte - An 8-bit quantity that may appear at any address in memory. Its bits are numbered from right to left starting with 0, the least significant bit (LSB), and ending with 7, the most significant bit (MSB).

halfword - A 16-bit quantity that may appear at any address in memory divisible by 2. Its bits are numbered from right to left starting with 0, the LSB, and ending with 15, the MSB.

word - A 32-bit quantity that may appear at any address in memory divisible by 4. Its bits are numbered from right to left starting with 0, the LSB, and ending with 31, the MSB.

Each of these types may be interpreted as a signed or unsigned quantity. A signed quantity is represented in 2's complement form. Therefore, for a signed quantity, the MSB indicates the sign of the quantity; 0 for a positive quantity and 1 for a negative quantity.

A *bit field* is a sequence of 1 to 32 bits contained in a base word. The field is specified by the address of its base word, a bit offset, and a width. The *bit offset* ranges from 0 to 31

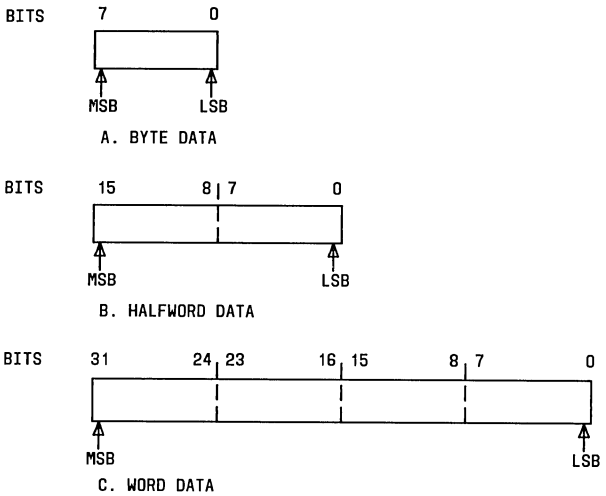


Figure 3-1. Bit Order of Data

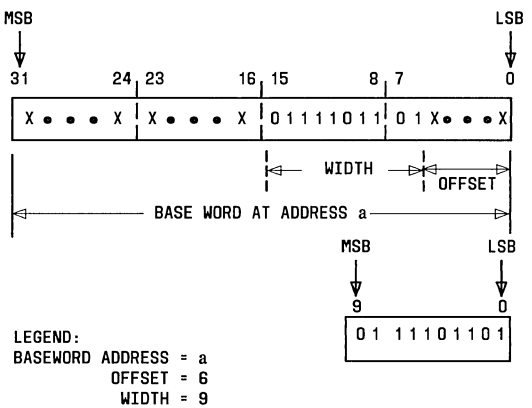


Figure 3-2. Bit Order in a Bit Field

and identifies the starting bit of the field. The offset count starts at the LSB of the base word and corresponds to the number of the bit in the word. That bit becomes bit 0, the LSB of the field. The *width* ranges from 0 to 31 and specifies the size of the field. Width plus one is the number of bits in the field. The width is numbered from right to left in the field and corresponds to the bit number of the field's MSB. Fields do not extend across word boundaries. Fields wrap around from MSB to LSB at the word boundary. Figure 3-2 illustrates a bit field located at address **a**, with an offset of 6, and a width of 9. Notice that the field contains 10 bits, one bit more than the width.

3.1.1 Sign and Zero Extension

All operations are performed only on 32-bit quantities even though an instruction may specify a byte or halfword operand. The *WE* 32100 Microprocessor reads in the correct number of bits for the operand and extends the data automatically to 32 bits. It uses *sign extension* when reading signed data or halfwords and *zero extension* when reading unsigned data or bytes (or bit fields that contain less than 32 bits). The data type of the source operand determines how many bits are fetched and what type of extension is applied. Bytes are treated as unsigned, while halfwords and words are considered signed. The type of extension applied can be changed using the expanded-operand type mode as described in **3.4.5 Expanded-Operand Type Mode**. For sign extension, the value of the MSB or sign bit of the data fills the high-order bits to form a 32-bit value. In zero extension, zeros fill the high order bits. The microprocessor automatically extends a byte or halfword to 32 bits before performing an operation. Figure 3-3 illustrates sign and zero extension.

An arithmetic, logical, data transfer, or bit field operation always yields an intermediate result that is 32 bits in length. If the result is to be stored in a register, the processor writes all 32 bits to that register. The processor automatically strips any surplus high-order bits from a result when writing bytes or halfwords to memory.

3.2 REGISTERS

A processor register may contain the operand for an instruction or may be used when computing an address of an operand. Therefore, most address modes, other than absolute, immediate, or literal, reference a processor register. In general, any of the sixteen processor registers may be used as an operand in all of the address modes. Table 3-1 lists the registers and assigned functions.

General-purpose registers r0 through r8 may be used for accumulation, addressing, or temporary data storage. The remaining processor registers are special purpose and are usually referenced with different names. Three of these registers are pointers to data stored on an execution stack: the frame pointer (FP), register 9 (r9), the argument pointer (AP), register 10 (r10), and the stack pointer (SP), register 12 (r12). Function calls and returns affect the AP, FP, and SP implicitly. The FP identifies the starting location of local variables for the function, while the AP identifies the beginning of the set of arguments passed to the function. The SP always points to the next available word location on the stack. Note that the stack grows upward to higher memory addresses.

Table 3-1. Register Set

Register	Name	Assembler Syntax	Assigned Function
0	r0	%r0	General-purpose (Note 1)
1	r1	%r1	General-purpose (Note 1)
2	r2	%r2	General-purpose (Note 1)
3	r3	%r3	General-purpose
4	r4	%r4	General-purpose
5	r5	%r5	General-purpose
6	r6	%r6	General-purpose
7	r7	%r7	General-purpose
8	r8	%r8	General-purpose
9	FP	%fp or %r9	Frame pointer
10	AP	%ap or %r10	Argument pointer
11	PSW	%psw or %r11	Processor status word (Note 2)
12	SP	%sp or %r12	Stack pointer
13	PCBP	%pcbp or %r13	Processor control block pointer (Note 2)
14	ISP	%isp or %r14	Interrupt stack pointer (Note 2)
15	PC	%pc or %r15	Program counter (Note 3)

Notes:
1. Block or string instructions may use this register as an implied argument for indexing or addressing. Operating system instructions also use these registers.
2. Privileged register. Writing to this register when the processor is not in kernel execution level causes a privileged-register exception (see **4.2.1 Execution Privilege**).
3. Registers 11 and 15 may not be used in some address modes (see **3.4 Address Modes**).

Some of the registers have restrictions on usage in instructions. Because registers 11, 13, and 14 (r11, r13, and r14) are privileged, these may be written only when kernel execution level is in effect. Register 11, the processor status word (PSW), contains status information about the current instruction and process. Register 13, the process control block pointer (PCBP), identifies a block of status information and pointers for a process. Register 14, the interrupt stack pointer (ISP), functions as a stack pointer for the interrupt stack.

The last register is the program counter, register 15 (r15). This register and register 11 may not be referenced in some address modes (see **3.4 Address Modes**). In addition, it is referenced implicitly in all program-control instructions and for all function calls and returns.

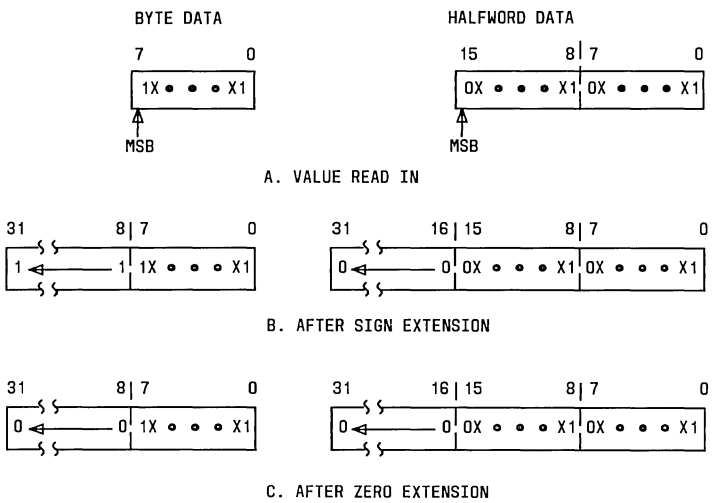


Figure 3-3. Extending Data to 32 Bits

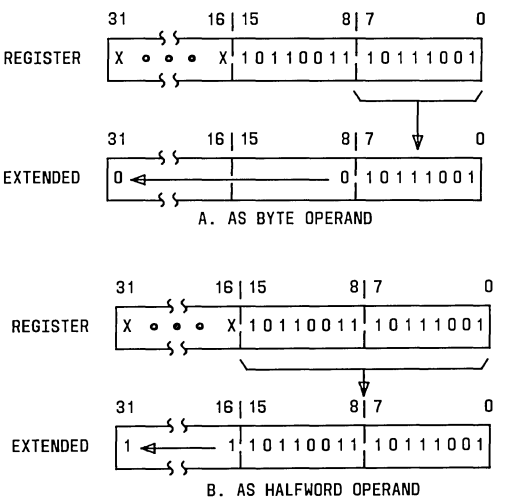


Figure 3-4. Register as a Source Operand

3.2.1 Writing and Reading Registers

A write to a register always affects all 32 bits. When a destination operand is a register, the processor ignores the data type of the operand and copies all 32 bits of a result to that register.

When reading from a register, the data type of the source operand determines how many bits are fetched and what type of extension is applied (see Figure 3-4). If a register is a byte operand, bits 0 through 7 of the register are fetched, and *zero* extension produces the 32-bit value required internally. If a register is halfword operand, bits 0 through 15 are fetched, but *sign* extension forms the 32-bit value.

3.3 INSTRUCTION FORMAT

Instructions may appear at any byte address. An instruction consists of a one- or two-byte opcode followed by zero or up to four operands. In assembly language, the mnemonic replaces the opcode and is followed by its operands. This is represented as

mnemonic opnd1,opnd2,...,opnd4

where the mnemonic is separated from the operands by a white space and commas are used to separate operands.

Part A of Figure 3-5 shows the general format of an instruction in memory. Each operand may consist of a descriptor byte followed by up to four bytes of embedded data. Part B of Figure 3-5 shows the general format of the operand. During execution, the program counter always points to the starting address (opcode byte) of the instruction.

3.3.1 Data Embedded in Operands

Figure 3-6 illustrates the format for operands with embedded word, halfword, and byte data. The first byte is the *operand descriptor* that defines which address mode and register the operand uses. The descriptor is divided into two 4-bit fields. Bits 0 through 3 define the register field; bits 4 through 7 define the address mode. The register field and address mode combinations are shown in Table 3-2.

There are two cases of operands with embedded data that do not have operand descriptors. First, when the operand is used as a target in a branch instruction, the operand is used as an 8- or 16-bit displacement from the program counter and no descriptor is needed. Second, there is no descriptor when a command word appears in a coprocessor instruction.

3.4 ADDRESS MODES

The *WE* 32100 Microprocessor recognizes the commonly used address modes such as immediate, register, absolute, and displacement or offset from the content of a register. Some modes involve a *pointer*, the address of a word location in memory that contains the address of the operand, and are known as *deferred* modes.

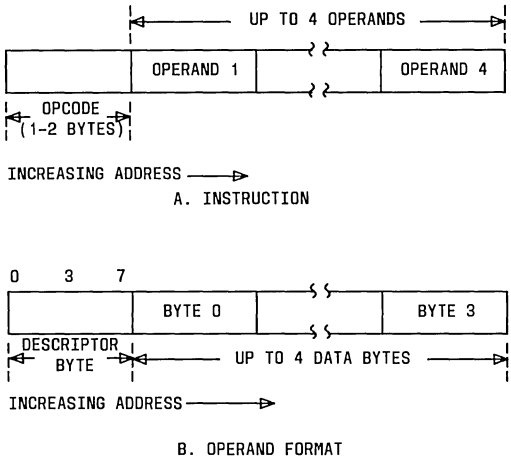


Figure 3-5. General Instruction Format

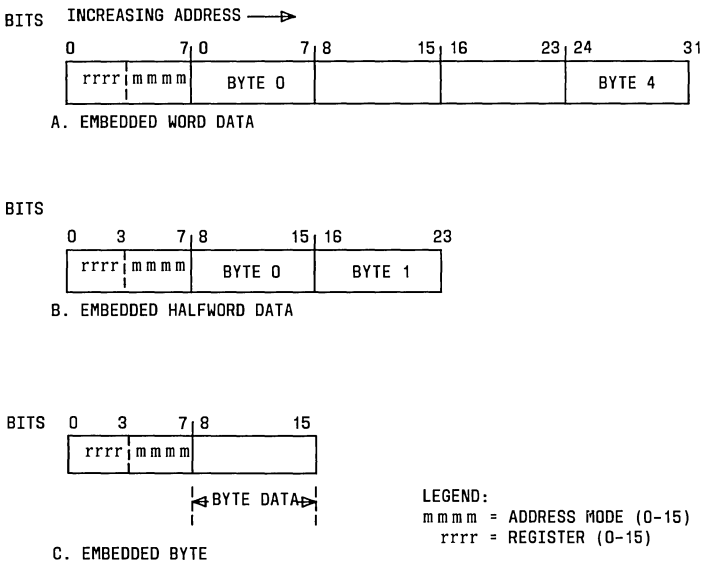


Figure 3-6. Data Embedded in an Operand

In assembly language, the syntax of the operand defines the operand and its address mode. Each address mode description in this section includes an example using a move instruction (MOVB, MOVH, or MOVW) to be described later. Because each example includes two operands, only the first operand demonstrates the address mode being described. The second operand uses the register mode.

Table 3-2 lists the address modes and gives the syntax for each. The descriptions and the table use the following notation:

0xnnn	Hexadecimal number nnn, where n is a hexadecimal digit 0 to 9 or a to f (or A to F); may also be written 0Xnnn
ap	Argument pointer (AP); contains the starting location on the stack of a list of arguments for a function
expr	User-supplied expression that yields a byte, halfword, or word
fp	Frame pointer (FP); contains the starting location on the stack of local variables for a function
imm8	Signed integer in the range −128 to +127 (i.e., -2^7 to $+2^7-1$)
imm16	Signed integer in the range −32768 to +32767; i.e., -2^{15} to $(+2^{15}-1)$
imm32	Signed integer in the range -2^{31} to $(+2^{31}-1)$
lit	Signed integer in the range −16 to +63
opnd	An operand that uses a mode other than the expanded-operand type
%rn	References a processor register; use the syntax shown in Table 3-1 for the desired register
so	Short offset; an integer in the range 0 to 14
type	Data type: sbyte (for signed byte), byte or ubyte (for unsigned byte), half or shalf (for signed halfword), uhalf (for unsigned halfword), word or sword (for signed word), uword (for unsigned word); see **3.4.5 Expanded-Operand Type Mode** for more details.

In machine language, a descriptor defines all source or destination operands and occupies one or more bytes in the instruction stream.

The first byte of the operand, called the *descriptor byte*, defines the address mode. (The expanded-operand type mode uses two descriptor bytes and is discussed later in this section.) Bytes that follow the descriptor byte contain any data required by the address mode for that operand. Table 3-2 identifies the total bytes in memory required for each mode.

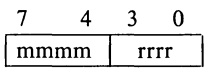
Table 3-2. Addressing Modes

Mode	Syntax	Mode Field	Register Field	Total Bytes	Notes
Absolute					
Absolute	$*expr*	7	15	5	—
Absolute deferred	*$*expr*	14	15	5	—
Displacement (from a register)					
Byte displacement	*expr*(%r*n*)	12	0—10,12—15	2	—
Byte displacement deferred	**expr*(%r*n*)	13	0—10,12—15	2	—
Halfword displacement	*expr*(%r*n*)	10	0—10,12—15	3	—
Halfword displacement deferred	**expr*(%r*n*)	11	0—10,12—15	3	—
Word displacement	*expr*(%r*n*)	8	0—10,12—15	5	—
Word displacement deferred	**expr*(%r*n*)	9	0—10,12—15	5	—
AP short offset	*so*(%ap)	7	0—14	1	1
FP short offset	*so*(%fp)	6	0—14	1	1
Immediate					
Byte immediate	*&imm8*	6	15	2	2,3
Halfword immediate	*&imm16*	5	15	3	2,3
Word immediate	*&imm32*	4	15	5	2,3
Positive literal	*&lit*	0—3	0—15	1	2,3
Negative literal	*&lit*	15	0—15	1	2,3
Register					
Register	%r*n*	4	0—14	1	1,3
Register deferred	(%r*n*)	5	0—10,12—14	1	1
Special Mode					
Expanded-operand type	{*type*}*opnd*	14	0—14	2—6	4

Notes:

1. Mode field has special meaning if register field is 15; see absolute or immediate mode.
2. Mode may not be used for a destination operand.
3. Mode may not be used if the instruction takes effective address of the operand.
4. *type* overrides instruction type; *type* determines the operand type, except that it does not determine the length for immediates or literals or whether literals are signed or unsigned. *opnd* determines actual address mode. For total bytes, add 1 to byte count for address mode determined by *opnd*.

As described before, the descriptor byte has two 4-bit fields:



The register field **rrrr**, bits 0 through 3, contains the number of a register, 0 through 15. The mode field **mmmm**, bits 4 through 7, contains an address-mode number, 0 through 15. Table 3-2 lists the value in the mode field and the possible values in the register field for each address mode. If the register field contains 15, the mode field may be interpreted differently.

In the following examples for the address modes, the first operand illustrates the mode while the second operand uses the register mode. For assembly language programming, values follow the C language conventions:

- Leading 0x or 0X denotes a hexadecimal value
- Leading 0 followed by the digits 0 through 7 is octal
- Digits 0 through 9, but no leading zero is decimal.

The byte boxes illustrating the instruction stream in the following examples contain hexadecimal values.

3.4.1 Absolute Address Modes

In this mode, an absolute address is embedded in the operand. This may be the address of the operand or of a pointer.

Absolute

The operand is accessed by an absolute address computed from the expression *expr*.

Syntax: *$expr*

mmmm: 7
rrrr: 15
Total bytes: 5

Example: MOVB $0x100,%r0

87	Opcode
7F	First Operand
00	
01	
00	
00	
40	Second Operand

This instruction moves the byte at location 100 to register 0(r0). %r0 is the syntax for the register mode. In the instruction stream, the four bytes that follow the descriptor byte form the 32-bit absolute address of the operand. The bytes follow the order shown on Figure 3-6 for word data.

Absolute Deferred

The operand is accessed through the absolute address of a *pointer*, a location in general memory that contains the address of the operand. The absolute address of this pointer is computed from the expression *expr*.

Syntax: **$expr*

mmmm: 14
rrrr: 15
Total bytes: 5

Example: MOVB *$0x2E00,%r1

87	Opcode
EF	First Operand
00	
2E	
00	
00	
41	Second Operand

This example moves a byte from memory to register 1 (r1). However, it uses a pointer (the word starting at location 0x2E00) to locate the byte in memory. In the instruction stream, the four bytes that follow the descriptor byte form the 32-bit absolute address of a word location in memory. That location contains the address of the operand. The 32-bit absolute address in the instruction follows the byte order shown on Figure 3-6 for word data.

3.4.2 Displacement Modes

For these modes, a displacement contained in the operand added to a register forms the address of the operand or a pointer to the operand. Sign-extension expands the displacement of 32 bits before the addition occurs.

Byte Displacement

A byte displacement added to a register forms the address of the operand. The displacement, computed from the expression *expr*, ranges from −128 to +127, and *n* ranges from 0 to 10 and 12 to 15 (use the syntax given in Table 3-1).

Syntax: *expr*(%r*n*)

mmmm: 12
rrrr: 0 to 10, 12 to 15
Total bytes: 2

Example: MOVB 6(%r1),%r0

87	Opcode
C1	First Operand
06	
40	Second Operand

This example moves a byte from memory to register 0. This byte in memory is located by adding the displacement 6 to register 1. The displacement is the byte that follows the descriptor byte in the instruction stream. This displacement is sign extended and added to the contents of the register 1. The sum is the address of the operand.

Byte Displacement Deferred

A byte displacement added to a register forms a pointer. The word location identified by the pointer contains the address of the operand. The displacement computed from the expression *expr* ranges from −128 to +127, and *n* ranges from 0 to 10 and 12 to 15 (use the syntax given in Table 3-1).

Syntax: **expr*(%r*n*)

mmmm: 13
rrrr: 0 to 10, 12 to 15
Total bytes: 2

Example: MOVB*0x30(%r2),%r3

87	Opcode
D2	First Operand
30	
43	Second Operand

This example adds the byte displacement 0x30 to the contents of register 2 (r2) to form the starting address of a pointer in memory. The pointer is the address of a byte in memory. After zero extension of the byte, the value is written to register 3 (r3). The displacement is the byte that follows the descriptor byte in the instruction stream. This byte is sign extended and added to the contents of register 2. The sum is the address of a word location in memory that contains the address of the operand.

Halfword Displacement

A halfword displacement added to a register forms the address of the operand. The displacement is computed from the expression *expr* and ranges from -2^{15} to $(+2^{15}-1)$.

Syntax: *expr*(%r*n*)

mmmm: 10
rrrr: 0 to 10, 12 to 15
Total bytes: 3

Example: MOVB 0x1101(%r2),%r8

87	Opcode
A2	First Operand
01	
11	
48	Second Operand

This example adds the halfword displacement 0x1101 to the contents of register 2. The result is the address of a byte in memory. This byte is written to register 8 after zero extension. In the instruction stream, the halfword that follows the descriptor byte is the displacement. This displacement is sign extended and added to the contents of register 2. The sum is the address of the operand. The displacement stored in the instruction follows the byte ordering shown on Figure 3-6 for halfword data.

Halfword Displacement Deferred

A halfword displacement added to a register *n* forms a pointer. The word location identified by the pointer contains the address of the operand. The displacement computed from the expression *expr* ranges from -2^{15} to $(+2^{15}-1)$, and *n* ranges from 0 to 10 and 12 to 15 (use the syntax given in Table 3-1).

Syntax: **expr*(%r*n*)

mmmm: 11
rrrr: 0 to 10, 12 to 15
Total bytes: 3

Example: MOVB *0x200(%r2),%r6

87	Opcode
B2	First Operand
00	
02	
46	Second Operand

This instruction adds the halfword displacement 0x200 to the contents of register 2, forming the address that locates a pointer in memory. The pointer locates a byte in memory that is written to register 6 after zero extension. In the instruction stream, the halfword that follows the descriptor byte is the displacement. This displacement is sign extended and added to the contents of register 2. The sum is the address of a word location in memory that contains the address of the operand. The displacement in the instruction stream follows the byte order shown on Figure 3-6 for halfword data.

Word Displacement

A word displacement added to a register forms the address of the operand. The displacement computed from the expression *expr* ranges from -2^{31} to $(+2^{31}-1)$, and *n* ranges from 0 to 10 and 12 to 15 (use the syntax given in Table 3-1).

Syntax: *expr*(%r*n*)

mmmm: 8
rrrr: 0 to 10, 12 to 15
Total bytes: 5

Example: MOVB 0x112234(%r2),%r4

87	Opcode
82	First Operand
34	
22	
11	
00	
44	Second Operand

The word displacement 0x112234 added to the contents of register 2 forms the address of a byte. The byte is stored in register 4 (r4) after zero extension. In the instruction stream, the byte that follows the descriptor byte is the displacement. This displacement is sign extended and added to the contents of the register 2. The sum is the address of the operand. The displacement stored in the instruction follows the byte ordering shown on Figure 3-6 for word data.

Word Displacement Deferred

A word displacement added to a register forms the address of a pointer. The pointer is the address of the operand in memory. The displacement computed from the expression *expr* ranges from -2^{31} to $+2^{31}-1$, and *n* ranges from 0 to 10 and 12 to 15 (use the syntax given in Table 3-1).

Syntax: **expr*(%r*n*)

mmmm: 9
rrrr: 0 to 10, 12 to 15
Total bytes: 5

Example: MOVB *0x20304050(%r2),%r0

87	Opcode
92	First Operand
50	
40	
30	
20	
40	Second Operand

The word displacement 0x20304050 added to the contents of register 2 forms an address of a pointer in memory. That pointer identifies the location of a byte to be written to register 0 after zero extension. In the instruction stream, the word that follows the descriptor byte is the displacement. This displacement is sign extended and added to the contents of register 2. The sum is the address of a word location in memory that contains the address of the operand. The displacement in the instruction stream follows the byte order shown on Figure 3-6 for word data.

AP Short Offset

This mode applies a short offset to the argument pointer (referenced as %ap) to locate an argument to a function. The offset *so* ranges from 0 through 14 and is added to AP to form the address of the argument.

Syntax: *so*(%ap)

mmmm: 7
rrrr: 0 through 14 (see text that follows)
Total bytes: 1

Example: MOVW 4(%ap),%r3

84	Opcode
74	First Operand
43	Second Operand

The offset 4 added to the contents of AP locates a word that is written to register 3. In the instruction stream, the 4-bit register field serves as the offset (a literal ranging from 0 through 14). This offset is sign extended and added to the contents of AP to locate a word, or argument, on the stack.

FP Short Offset

This mode applies a short offset to the frame pointer, referenced as %fp, to locate a local variable for a function. The offset *so* ranges from 0 through 14 and is added to FP to form the address of the variable.

Syntax: *so*(%fp)

mmmm: 6
rrrr: 0 through 14 (see text that follows)
Total bytes: 1

Example: MOVW 12(%fp),%r0

84	Opcode
6C	First Operand
40	Second Operand

The offset 12 added to the contents of FP locates a word (a local variable) that is written to register 0. In the instruction stream, the 4-bit register field serves as the offset (a literal ranging from 0 through 14). This offset is sign extended and added to the contents of FP.

3.4.3 Immediate Modes

For these modes, the instruction stream contains the operand data. The type of the mnemonic does not affect the width of an operand that uses these address modes.

Byte Immediate

The operand is the signed 8-bit immediate value *imm8* that ranges from −128 to +127.

Note: This address mode may not be used as a destination or for an effective address. Either usage causes an illegal-operand exception.

Syntax: *&imm8*

mmmm: 6
rrrr: 15
Total bytes: 2

Example: MOVW &40,%r6

84	Opcode
6F	First Operand
28	
46	Second Operand

The byte value 40 replaces the contents of register 6. The mnemonic specifies a word operation, but the immediate value remains a byte. In the instruction stream, the byte that follows the descriptor byte contains an 8-bit immediate value that ranges from −128 to +127.

Halfword Immediate

The operand is the signed 16-bit immediate value *imm16* that ranges from -2^{15} to $(+2^{15}-1)$.

Note: This address mode may not be used as a destination or for an effective address. Either usage causes an illegal-operand exception.

Syntax: *&imm16*

mmmm: 5
rrrr: 15
Total bytes: 3

Example: MOVW &0x1234,%r2

84	Opcode
5F	First Operand
34	
12	
42	Second Operand

Here, the halfword value 0x1234 replaces the contents of register 2. In the instruction stream, the halfword that follows the descriptor byte contains a 16-bit immediate value that ranges from -2^{15} to $(+2^{15}-1)$. This immediate value is stored in the byte order shown on Figure 3-6 for halfword data.

Word Immediate

The operand is the signed 32-bit immediate value *imm32* that ranges from -2^{31} to $(+2^{31}-1)$.

Note: This address mode may not be used as a destination or for an effective address. Either usage causes an illegal-operand exception.

Syntax: *&imm32*

mmmm: 4
rrrr: 15
Total bytes: 5

Example: MOVW &0x12345678,%r3

84	Opcode
4F	First Operand
78	
56	
34	
12	
43	Second Operand

In this example, the word value 0x12345678 replaces the contents of register 3. In the instruction stream, the word that follows the descriptor byte contains a 32-bit immediate value that ranges from -2^{31} to $+2^{31}-1$. This immediate value is stored in the byte order shown on Figure 3-6 for word data.

Positive Literal

The operand is the unsigned 6-bit literal value *lit* that ranges from 0 to 63.

Note: This address mode may not be used as a destination or for an effective address. Either usage causes an illegal-operand exception.

Syntax: *&lit*

mmmm: 0 to 3
rrrr: 0 to 15
Total bytes: 1

Example: MOVB &4,%r4

87	Opcode
04	First Operand
44	Second Operand

Here, the positive literal 4 replaces the contents of register 4. Zeros fill the high-order bits in the register. In the instruction stream, the descriptor byte provides an unsigned 6-bit literal that ranges from 0 to 63. It is formed by concatenating the 4-bit register (rrrr) field with the two low-order bits of the mode (mmmm) field; i.e., bits 0 through 5 of the descriptor byte form the literal.

Negative Literal

The operand is the signed 8-bit literal value *lit* that ranges from −1 to −16.

Note: This address mode may not be used as a destination or for an effective address. Either usage causes an illegal-operand exception.

Syntax: *&lit*

mmmm: 15
rrrr: 0 to 15
Total bytes: 1

Example: MOVB &−1,%r0

87	Opcode
FF	First Operand
40	Second Operand

In the instruction stream, the descriptor byte provides a signed 8-bit literal that ranges from −1 to −16. It is formed by concatenating the 4-bit register (rrrr) field with the 4-bit mode (mmmm) field; i.e., the 8-bit descriptor byte forms the literal.

3.4.4 Register Modes

These modes use the contents of a register as the operand or as a pointer to the operand.

Register Mode

In this mode, the register *n*, which ranges from 0 to 14, is the operand.

Note: This mode may not be used if the opcode takes the effective address of the operand.

Syntax: *%rn*

mmmm: 4
rrrr: 0 to 14
Total bytes: 1

Example: MOVB %r0,%ap

87	Opcode
40	First Operand
4A	Second Operand

This example moves a byte from one register to another. It reads bits 0 through 7 of register 0, extends a zero through 32 bits, and writes the result to register 10, the argument pointer. In the instruction stream, the register specified in the register field is the operand.

Register Mode Deferred

The register *n*, which ranges from 0 to 10 and 12 to 14, contains a pointer to the operand.

Syntax: (%r*n*)

mmmm: 5
rrrr: 0 to 10, 12 to 14
Total bytes: 1

Example: MOVH (%r2),%r1

86	Opcode
52	First Operand
41	Second Operand

Here, register 2 contains the address of a halfword that is read. The halfword is sign extended through 32 bits, and the result is written to register 1. In the instruction stream, the register specified in the register field contains a pointer to a word location in memory that is the operand.

3.4.5 Expanded-Operand Type Mode

Normally, the opcode controls the type of all operands for the instruction. This mode changes the type of an operand and those that follow it in an instruction.

Note: The expanded-operand type mode does not affect the length of immediate operands, but does affect whether they are treated as signed or unsigned. The expanded-operand mode does not affect the treatment of literals.

In assembly language, the syntax of this mode is

{*type*}*opnd*

where *opnd* is an operand descriptor that uses any address mode other than the expanded-operand type mode.

When the expanded-operand type mode is used, *type* overrides the type for this operand, except as noted above, and *opnd* becomes the real address mode for the operand. The new type remains in effect for the operands that follow in the instruction unless another expanded-operand mode overrides it. Table 3-3 lists the syntax for *type*.

This mode requires two descriptor bytes (see Figure 3-7). The first byte identifies the expanded-operand mode and the new type, while the second is the descriptor byte for the address mode.

The type field *tttt* contains the value of the new type (see Table 3-3). The second byte contains the mode field (mmmm) and the register field (rrrr) for the address mode. This byte is the descriptor byte for the new address mode.

For example, the following instruction converts a signed byte into an unsigned halfword:

MOVB {sbyte}%r0,{uhalf}4(%r1)

The first operand's real mode is register, the second operand is byte displacement. The instruction reads bits 0 through 7 from register 0, extends the sign through 32 bits, and writes an unsigned halfword. In the instruction stream, the bytes contain the following:

87	Opcode
E7	First Operand
40	
E2	Second Operand
C1	
04	

Note: Expanded-operand type mode is illegal with coprocessor instructions with operands CALL, SAVE, RESTORE, SWAP INTERLOCKED, PUSHW, PUSHAW, POPW, and JSB instructions and will generate an illegal operand fault.

Table 3-3. Options for *type* in Expanded-Operand Mode

Type	Syntax	tttt Field (See Note)
Signed byte	**sbyte**	7
Signed halfword	**half** or **shalf**	6
Signed word	**word** or **sword**	4
Unsigned byte	**byte** or **ubyte**	3
Unsigned halfword	**uhalf**	2
Unsigned word	**uword**	0

Note: Types are not defined for the values 1, 5, and 8 through 14; using these generates a reserved-data-type exception.

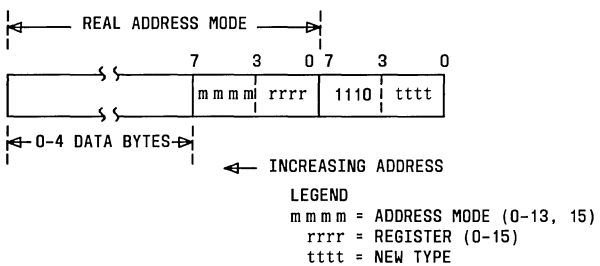


Figure 3-7. Expanded-Operand Type Descriptor

3.5 CONDITION FLAGS

Bits 21 to 18 of the processor status word (PSW) contain four condition flags (N, Z, V, and C) that are set by most instructions. The order is shown on Figure 3-8. The conditional program-control instructions check one or more of these flags before executing the branch, jump, or return. In general, these flags reflect the result of the most recent instruction that affects them. Most instructions set the flags according to standard criteria. Before defining that criteria, the following terms are defined:

- *Result* refers to the internal result of the operation as if it were performed in an infinite-precision machine. The microprocessor operates on 32-bit data internally and uses a 33-bit space for the internal result. Bytes and halfwords read in are extended to 32 bits before the operation. The destination operand determines the *type* (i.e., signed or unsigned, and size: byte, halfword, or word) of this result.
- *Output value* refers to the data written to the destination location. The size of this data, 8, 16, or 32 bits, corresponds to the data type of the destination operand: byte, halfword, or word, respectively.

The following conditions cause the appropriate flag bit to be altered:

N *Negative* (PSW bit 21) - Logical instructions change N to the setting of the output value of the MSB: bit 31 for words, bit 15 for halfwords, and bit 7 for bytes. For all other instructions, N is set if the sign of the result is negative. If truncation occurs, the N flag may be set even though the sign bit of the output value is zero. Zero is considered positive.

Z *Zero* (PSW bit 20) - Logical instructions set Z if the output value is zero. For all other instructions Z is set if the result is equal to zero. If truncation occurs, the Z flag may not be set even though all bits of the output value are zero.

V *Overflow* (PSW bit 19) - For instructions with a signed destination, V is set if the sign bit of the output value is different from any truncated bit of the result. For instructions with an unsigned destination, V is set if any truncated bit is a 1. The arithmetic left shift operation sets the V bit only if a truncation error occurs. Bit, compare, and test instructions always reset V.

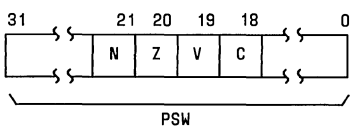


Figure 3-8. Condition Flags

C *Carry/Borrow* (PSW bit 18) - Logical instructions clear this bit. For all other instructions, the type of the result determines the state of the C bit. C is set if a *carry* occurs into the 33rd bit for word operations, into the 17th bit for halfword operations, or into the 9th bit for byte operations. The C bit is set if a *borrow* occurs from these bits for subtract, negate, and decrement. For example, consider A minus B where A and B are unsigned. If A $\geqslant$ B after both are extended to 32 bits, then C is cleared. Otherwise, the C flag is set.

Note: If a memory-write fault occurs, the flags are set as if the instruction was completed normally.

The instruction descriptions later in this chapter include the effect that each instruction has on the condition flags.

3.6 FUNCTIONAL GROUPS

The *WE* 32100 Microprocessor instruction set may be separated into six functional groups: data transfer instructions, arithmetic instructions, logical instructions, program control instructions, coprocessor instructions, and stack and miscellaneous instructions. This section contains a description of each group, along with an instruction listing of each group. The conditions column in the instruction listing refers to the condition flag code assignment cases listed in Table 3-10. (For more details of individual instructions see **3.7 INSTRUCTION SET LISTINGS.**)

3.6.1 Data Transfer Instructions

These instructions transfer data to and from registers and memory. Most of them have three types (indicated by the last character of the mnemonic): byte (B), halfword (H), and word (W). A mnemonic's type determines the type of each operand in the instruction, unless the expanded-operand type mode changes an operand's type. The type of the destination operand (*dst*) determines how the condition flags are set (see **3.5 CONDITION FLAGS**). The instructions have a read-only source operand (*src*) and a read/write destination operand.

Table 3-4. Data Transfer Instruction Group

Instruction	Mnemonic	Opcode	Conditions*
Move:			
Move byte	MOVB	0x87	
Move halfword	MOVH	0x86	
Move word	MOVW	0x84	
Move address (word)	MOVAW	0x04	Case 1
Move complemented byte	MCOMB	0x8B	
Move complemented halfword	MCOMH	0x8A	
Move complemented word	MCOMW	0x88	
Move negated byte	MNEGB	0x8F	
Move negated halfword	MNEGH	0x8E	Case 2
Move negated word	MNEGW	0x8C	
Move version number	MVERNO	0x3009	Unchanged
Swap (Interlocked):			
Swap byte interlocked	SWAPBI	0x1F	
Swap halfword interlocked	SWAPHI	0x1E	Case 1
Swap word interlocked	SWAPWI	0x1C	
Block Operations:			
Move block of words	MOVBLW	0x3019	Unchanged
Field Operations:			
Extract field byte	EXTFB	0xCF	
Extract field halfword	EXTFH	0xCE	
Extract field word	EXTFW	0xCC	Case 1
Insert field byte	INSFB	0xCB	
Insert field halfword	INSFH	0xCA	
Insert field word	INSFW	0xC8	
String Operations:			
String copy	STRCPY	0x3035	Unchanged
String end	STREND	0x301F	

* Refer to Table 3-10 for condition flag code assignments.

3.6.2 Arithmetic Instructions

Arithmetic instructions perform arithmetic operations on data in registers and memory. Most of these instructions have three types (specified by the last character of the mnemonic): byte (B), halfword (H), and word (W). This type specification applies to each operand in the instruction, unless the expanded-operand type mode changes an operand's type. The type of the destination operand (*dst*) determines how the condition flags are set (see **3.5 CONDITION FLAGS**).

Many arithmetic operations are available as two- or three-address instructions. A two-address instruction has a read-only source operand (*src*) and a read/write destination operand. Three-address instructions have two read-only source operands (*src1, src2*) and a write-only destination operand. A few instructions also have a read-only count operand (*count*).

If the result of an arithmetic operation is too large to be represented in 32 bits, the high-order bits are truncated and the processor issues an integer-overflow exception.

Table 3-5. Arithmetic Instruction Group

Instruction	Mnemonic	Opcode	Conditions*
Add:			
Add byte	ADDB2	0x9F	
Add halfword	ADDH2	0x9E	
Add word	ADDW2	0x9C	
Add byte, 3-address	ADDB3	0xDF	
Add halfword, 3-address	ADDH3	0xDE	
Add word, 3-address	ADDW3	0xDC	
Subtract:			
Subtract byte	SUBB2	0xBF	
Subtract halfword	SUBH2	0xBE	Case 2
Subtract word	SUBW2	0xBC	
Subtract byte, 3-address	SUBB3	0xFF	
Subtract halfword, 3-address	SUBH3	0xFE	
Subtract word, 3-address	SUBW3	0xFC	
Increment:			
Increment byte	INCB	0x93	
Increment halfword	INCH	0x92	
Increment word	INCW	0x90	
Decrement:			
Decrement byte	DECB	0x97	
Decrement halfword	DECH	0x96	
Decrement word	DECW	0x94	

* Refer to Table 3-10 for condition flag code assignments.

Table 3-5. Arithmetic Instruction Group (Continued)			
Instruction	**Mnemonic**	**Opcode**	**Conditions***
Multiply:			
Multiply byte	MULB2	0xAB	
Multiply halfword	MULH2	0xAA	Case 3
Multiply word	MULW2	0xA8	
Multiply byte, 3-address	MULB3	0xEB	
Multiply halfword, 3-address	MULH3	0xEA	Case 4
Multiply word, 3-address	MULW3	0xE8	
Divide:			
Divide byte	DIVB2	0xAF	
Divide halfword	DIVH2	0xAE	Case 3
Divide word	DIVW2	0xAC	
Divide byte, 3-address	DIVB3	0xEF	
Divide halfword, 3-address	DIVH3	0xEE	Case 4
Divide word, 3-address	DIVW3	0xEC	
Modulo:			
Modulo byte	MODB2	0xA7	
Modulo halfword	MODH2	0xA6	Case 3
Modulo word	MODW2	0xA4	
Modulo byte, 3-address	MODB3	0xE7	
Modulo halfword, 3-address	MODH3	0xE6	Case 4
Modulo word, 3-address	MODW3	0xE4	
Arithmetic Shift:			
Arithmetic left shift word	ALSW3	0xC0	Case 5
Arithmetic right shift byte	ARSB3	0xC7	
Arithmetic right shift halfword	ARSH3	0xC6	Case 3
Arithmetic right shift word	ARSW3	0xC4	

* Refer to Table 3-10 for condition flag code assignments.

3.6.3 Logical Instructions

Logical instructions perform logical operations on data in registers and memory. Most of these instructions have three types (specified by the last character of the mnemonic): byte (B), halfword (H), and word (W). A mnemonic's type determines the type of each operand in the instruction, unless the expanded-operand type mode changes an operand's type. The type of the destination operand (*dst*) determines how the condition flags are set (see **3.5 CONDITION FLAGS**).

Many logical operations are available as two- or three-address instructions. A two-address instruction has a read-only source operand (*src*) and a read/write destination operand (*dst*). Three-address instructions have two read-only source operands (*src1, src2*) and a write-only destination operand. A few instructions have a read-only count operand (*count*).

Table 3-6. Logical Group

Instruction	Mnemonic	Opcode	Conditions*
AND:			Case 1
AND byte	ANDB2	0xBB	
AND halfword	ANDH2	0xBA	
AND word	ANDW2	0xB8	
AND byte, 3-address	ANDB3	0xFB	
AND halfword, 3-address	ANDH3	0xFA	
AND word, 3-address	ANDW3	0xF8	
Exclusive OR (XOR):			
Exclusive OR byte	XORB2	0xB7	
Exclusive OR halfword	XORH2	0xB6	
Exclusive OR word	XORW2	0xB4	
Exclusive OR byte, 3-address	XORB3	0xF7	
Exclusive OR halfword, 3-address	XORH3	0xF6	
Exclusive OR word, 3-address	XORW3	0xF4	
OR:			
OR byte	ORB2	0xB3	
OR halfword	ORH2	0xB2	
OR word	ORW2	0xB0	
OR byte, 3-address	ORB3	0xF3	
OR halfword, 3-address	ORH2	OxF2	
OR word, 3-address	ORW3	OxF0	
Compare or Test:			Case 2
Compare byte	CMPB	0x3F	
Compare halfword	CMPH	0x3E	
Compare word	CMPW	0x3C	
Test byte	TSTB	0x2B	Case 6
Test halfword	TSTH	0x2A	
Test word	TSTW	0x28	
Bit test byte	BITB	0x3B	Case 1
Bit test halfword	BITH	0x3A	
Bit test word	BITW	0x38	
Clear:			Case 2
Clear byte	CLRB	0x83	
Clear halfword	CLRH	0x82	
Clear word	CLRW	0x80	
Rotate or Logical Shift:			Case 1
Rotate word	ROTW	0xD8	
Logical left shift byte	LLSB3	0xD3	
Logical left shift halfword	LLSH3	0xD2	
Logical left shift word	LLSW3	0xD0	
Logical right shift word	LRSW3	0xD4	

* Refer to Table 3-10 for condition flag code assignments.

3.6.4 Program Control Instructions

Program control instructions change the program sequence, but generally do not alter the condition flags.

Branch instructions have two types specified by the last character of the mnemonic: byte displacement (B) and halfword displacement (H). A mnemonic's type determines if an 8- or a 16-bit displacement is embedded in the instruction. This displacement (*disp8, disp16*) is read, its sign is extended through 32 bits, and the result is added to the program counter (PC) to compute the target address. Jump instructions have a read-only, 32-bit destination (*dst*) operand that replaces the contents of the PC.

Jump instructions are always unconditional, but both conditional and unconditional branch and return instructions are provided. Unconditional transfers change the contents of the PC to the value specified. Conditional transfers first examine the status of the processor's condition flags to determine if the transfer should be executed.

Subroutine and procedure-call (function) transfer instructions save or restore registers so execution can transfer to the subroutine or function and then return to the original program sequence.

Subroutine Transfer. A subroutine transfer is different from a normal transfer. Before transferring to a subroutine, it is necessary to save the address of the next instruction.

Branch, jump, and return instructions for subroutines always implicitly affect the stack pointer (SP). For subroutines, branch and jump save the address of the next instruction on the stack at the location identified by the SP, increment the SP by 4, and then alter the PC. Return from subroutine decrements the SP by 4, retrieves the saved address from the stack, and writes it to the PC.

Procedure Transfer. For procedure transfers it is necessary to save other registers. These instructions establish the environment for a function in a high-level language. Call and save instructions automatically save the calling function's pointers, set up pointers to the new function's environment, call the function, and save registers for local variables. Restore and return instructions remove that environment and return to the calling function.

A stack frame provides reserved space, including a register-save area, for each function. The register-save area stores the calling function's FP, AP, PC, and registers 3 through 8 (r3 through r8), if requested. Saving r3 through r8 gives the new function space for up to six register variables. The SP is not saved because its value is always implicit.

All function calls have a fixed-size register-save area, even though some of it may not be used. Save and restore control how many of the six user registers r3 through r8 will be saved and restored. A return from a function retrieves the saved pointers and registers to restore the original function's environment.

Table 3-7. Program Control Instructions

Instruction	Mnemonic	Opcode	Conditions
Unconditional Transfer:			
Branch with byte (8-bit) displacement	BRB	0x7B	
Branch with halfword (16-bit) displacement	BRH	0x7A	
Jump	JMP	0x24	
Conditional Transfers:			
Branch on carry clear byte	BCCB	0x53*	
Branch on carry clear halfword	BCCH	0x52*	
Branch on carry set byte	BCSB	0x5B*	
Branch on carry set halfword	BCSH	0x5A*	
Branch on overflow clear, byte displacement	BVCB	0x63	
Branch on overflow clear, halfword displacement	BVCH	0x62	
Branch on overflow set, byte displacement	BVSB	0x6B	
Branch on overflow set, halfword displacement	BVSH	0x6A	
Branch on equal byte (duplicate)	BEB	0x6F	
Branch on equal byte	BEB	0x7F	
Branch on equal halfword (duplicate)	BEH	0x6E	
Branch on equal halfword	BEH	0x7E	
Branch on not equal byte (duplicate)	BNEB	0x67	
Branch on not equal byte	BNEB	0x77	
Branch on not equal halfword (duplicate)	BNEH	0x66	Unchanged
Branch on not equal halfword	BNEH	0x76	
Branch on less than byte (signed)	BLB	0x4B	
Branch on less than halfword (signed)	BLH	0x4A	
Branch on less than byte (unsigned)	BLUB	0x5B*	
Branch on less than halfword (unsigned)	BLUH	0x5A*	
Branch on less than or equal byte (signed)	BLEB	0x4F	
Branch on less than or equal halfword (signed)	BLEH	0x4E	
Branch on less than or equal byte (unsigned)	BLEUB	0x5F	
Branch on less than or equal halfword (unsigned)	BLEUH	0x5E	
Branch on greater than byte (signed)	BGB	0x47	
Branch on greater than halfword (signed)	BGH	0x46	
Branch on greater than byte (unsigned)	BGUB	0x57	
Branch on greater than halfword (unsigned)	BGUH	0x56	
Branch on greater than or equal byte (signed)	BGEB	0x43	
Branch on greater than or equal halfword (signed)	BGEH	0x42	
Branch on greater than or equal byte (unsigned)	BGEUB	0x53*	
Branch on greater than or equal halfword (unsigned)	BGEUH	0x52*	
Return on carry clear	RCC	0x50*	
Return on carry set	RCS	0x58*	

* Indicates that opcode matches another instruction but operation is the same.

Table 3-7. Program Control Instructions (Continued)

Instruction	Mnemonic	Opcode	Conditions
Conditional Transfers (Continued):			
Return on overflow clear	RVC	0x60	
Return on overflow set	RVS	0x68	
Return on equal (unsigned)	REQLU	0x6C	
Return on equal (signed)	REQL	0x7C	
Return on not equal (unsigned)	RNEQU	0x64	
Return on not equal (signed)	RNEQ	0x74	
Return on less than (signed)	RLSS	0x48	
Return on less than (unsigned)	RLSSU	0x58*	
Return on less than or equal (signed)	RLEQ	0x4C	
Return on less than or equal (unsigned)	RLEQU	0x5C	Unchanged
Return on greater than (signed)	RGTR	0x44	
Return on greater than (unsigned)	RGTRU	0x54	
Return on greater than or equal (signed)	RGEQ	0x40	
Return on greater than or equal (unsigned)	RGEQU	0x50*	
Subroutine Transfer:			
Branch to subroutine, byte displacement	BSBB	0x37	
Branch to subroutine, halfword displacement	BSBH	0x36	
Jump to subroutine	JSB	0x34	
Return from subroutine	RSB	0x78	
Procedure Transfer:			
Save registers	SAVE	0x10	
Restore registers	RESTORE	0x18	
Call procedure	CALL	0x2C	
Return from procedure	RET	0x08	

* Indicates that opcode matches another instruction but operation is the same.

Program control instructions explicitly manipulate four registers:

1. PC - The call instruction saves the old PC as the return address (RA) and sets PC to the first executable instruction of the function being called. The return instruction restores PC to the RA (the next executable instruction of the calling function).
2. SP - These instructions adjust SP automatically to point to the top of the stack whenever they store or retrieve items.
3. FP - The save instruction sets FP to the address just above the saved registers. The FP accesses a region on the stack that stores temporary (or automatic) variables for the function.
4. AP - The call instruction adjusts AP to the beginning of a list of arguments for the function.

On a function call, the calling function contains a call instruction; the save instruction should be the first statement of the called function. For a return, a restore and a return appear in the function being exited.

Figure 3-9 shows the stack after the CALL-SAVE sequence:

```
        PUSHW arg1              /*push three arguments*/
        PUSHW arg2
        PUSHW arg3
        CALL -(3*4)(%sp),func1  /*call function*/
          .
          .                     /*other instructions*/
          .
func1:  SAVE %r3                /*save r3 through r8*/
```

First, three arguments are pushed onto the stack; each push increments SP. Then CALL automatically saves the old pointers. It uses its first operand to set AP to the beginning of the three arguments and its second operand to call the function. Next, SAVE, the first statement in the function, is executed, automatically saving registers r3 through r8. It also adjusts SP and FP for each push.

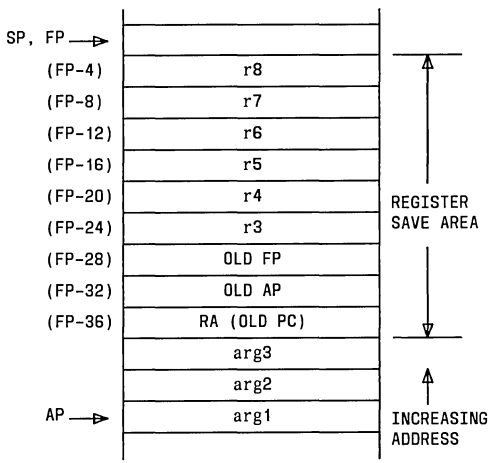


Figure 3-9. Stack After CALL-SAVE Sequence

To return to the original sequence, the function **func1** contains the following instructions:

```
func1:  SAVE %r3        /*save r3 through r8*/
          .
          .             /*other instructions*/
          .
        RESTORE %r3     /*restore r3 through r8*/
        RET             /*return to main function*/
```

The restore instruction retrieves registers r8 through r3 from the stack. It must have the same operand as the original SAVE; otherwise, the return (RET) cannot restore the correct AP and PC. Both instructions decrement SP as they pop the register contents from the stack.

3.6.5 Coprocessor Instructions

These instructions implement the interface with coprocessors. All coprocessor instructions have an 8-bit opcode followed by one word. This word is transmitted on the data bus and interpreted by the coprocessor. The word is not used by the CPU. If no coprocessor responds to the transmitted word, an external memory fault occurs.

After the word following the opcode is transmitted, the source operands, if any, are fetched from memory. The CPU then waits until the "coprocessor done" signal is asserted, after which the CPU attempts to read a word. If this access is faulted, an external memory fault occurs. If this access is not faulted, bits 18 through 21 of the word are copied into bits 18 through 21 (condition flags) of the PSW. The resulting operand, if any, is then written to memory.

Coprocessor instructions can have from zero to two operands. The operands may be of three data types (specified by the last character of the mnemonic): single-word (S), double-word (D), and triple-word (T). All operands must start on an address evenly divisible by four (a word boundary).

3.6.6 Stack and Miscellaneous Instructions

The stack instructions are used to manipulate the stack. The push and pop instructions always process a word and alter the SP. They have a read-only source operand *src* or a write-only destination operand *dst*.

Miscellaneous instructions include those that alter the machine state or have an effect on the cache memory. The breakpoint instruction causes a breakpoint-trap exception. Control transfers to the operating system for the appropriate exception handler. The NOP instructions come in three lengths: 1, 2, or 3 bytes. If an instruction, other than a conditional transfer, reads the PSW, the assembler **m32as** inserts a NOP before that instruction. This allows time for the PSW codes to settle before the new instruction tries to access them. Cache flush makes the instruction cache invalid.

Table 3-8. Coprocessor Instructions			
Instruction	**Mnemonic**	**Opcode**	**Conditions***
Coprocessor operation	SPOP	0x32	Case 10
Coprocessor operation read single	SPOPRS	0x22	
Coprocessor operation double	SPOPRD	0x02	
Coprocessor operation triple	SPOPRT	0x06	
Coprocessor operation single 2-address	SPOPS2	0x23	
Coprocessor operation double 2-address	SPOPD2	0x03	
Coprocessor operation triple 2-address	SPOPT2	0x07	
Coprocessor operation write single	SPOPWS	0x33	
Coprocessor operation write double	SPOPWD	0x13	
Coprocessor operation write triple	SPOPWT	0x17	

* Refer to Table 3-10 for condition flag code assignments.

Table 3-9. Stack and Miscellaneous Instructions			
Instruction	**Mnemonic**	**Opcode**	**Conditions***
Stack Operations:			Case 1
Push address word	PUSHAW	0xE0	
Push word	PUSHW	0xA0	
Pop word	POPW	0x20	
Miscellaneous:			Unchanged
No operation, 1 byte	NOP	0x70	
No operation, 2 byte	NOP2	0x73	
No operation, 3 byte	NOP3	0x72	
Breakpoint trap	BPT	0x2E	
Cache flush	CFLUSH	0x27	
Extended opcode	EXTOP	0x14	

* Refer to Table 3-10 for condition flag code assignments.

3.7 INSTRUCTION SET LISTINGS

Section **3.7.2 Instruction Set Descriptions** presents descriptions of each member of the instruction set for the *WE* 32100 Microprocessor. The descriptions are in alphabetical order, and any instructions that operate on more than one type of operand, byte, halfword, or word are listed on the same page. (For quick reference to the instructions by function, mnemonic, or opcode see Sections **3.7.3 Instruction Set Sumary by Fuction, 3.7.4 Instruction Set Summary by Mnemoic**, and **3.7.5 Instruction Set Summary by Opcode**.)

3.7.1 Notation

Each instruction description contains several parts: assembler syntax, opcode operation, address modes, condition flags, exceptions, examples, and notes (optional).

Assembler Syntax. Presents the assembly language syntax for the instruction, including any required spacing and punctuation. The user-specified elements appear in italics. All operands must appear in the order shown. If an instruction has byte, halfword, and word forms, all three forms are presented.

The syntax uses the following symbols to denote operands that may be written in the address modes shown in Table 3-2: *count, dst, offset, src, width.* Program control instructions use *disp8* or *disp16* as a displacement operand. The operand does not use an address mode, but is written as an 8- or 16-bit literal.

Opcodes. Lists each opcode with the appropriate mnemonic and function.

Operation. Describes the operation performed. The description generally uses C language syntax and the operators and symbols shown in Table 3-11.

Address Modes. Identifies the valid address modes for each operand. Refer to Table 3-2 for address mode syntax and to Table 3-1 for the syntax for referencing registers.

Condition Flags. Identifies the effect of the instruction on each of the condition flags.

Exceptions. Identifies any error conditions that may result in illegal operands, opcodes, or operations.

Examples. Presents examples of the instruction written in assembly language. In some cases, it will give the contents of registers before and after execution. Register bytes are read from right to left and their contents are given as hexadecimal values.

Notes (Optional). Explains other parts of the description when necessary.

Table 3-10. Condition Flag Code Assignments

Case	Condition Flags				Special Conditions*
	N(Negative)	Z(Zero)	C(Carry)	V(Overflow)	
1	MSB of *dst*	1 if *dst* = 0	0	0	V flag is set when expanded operand type mode is used, and the result is truncated when represented in destination.
2	1 if result < 0	1 if result = 0	1 on carry or borrow	1 on integer overflow	—

Table 3-10. Condition Flag Code Assignments (Continued)

Case	Condition Flags				Special Conditions*
	N(Negative)	Z(Zero)	C(Carry)	V(Overflow)	
3	1 if *dst* < 0	1 if *dst* = 0	0	1 on integer overflow	—
4	1 if *dst* < 0	1 if *dst* = 0	0	1 on integer overflow	V flag may not set when *dst* is signed word type, bit 31 of absolute value of the result is 1, and while bits 32—63 of the absolute value of the result are 0s.
5	1 if *dst* < 0	1 if *dst* = 0	0	0	V flag is set if expanded-operand type mode changes the type of *dst* and integer overflow occurs.
6	1 if *src* < 0	1 if *src* = 0	0	0	N flag is affected if *src* is signed integer.
7	MSB of word returned	1 if word returned = 0	0	0	—
8	—	—	—	—	All flags determined by new PSW.
9	—	—	—	—	All flags determined by restored PSW.
10	—	—	—	—	When coprocessor status word is accepted, bits 18—21 of the word read are put into bits 18—21 of the PSW, respectively.

Notes:

MSB - Most Significant Bit
dst - destination
src - source

* For cases 1 through 6, when the PSW is used as a source the condition flags are unaffected; when the PSW is used as a destination, the condition flags assume the value of bits 18—21 of the result of the operation performed.

Table 3-11. Assembly Language Operators and Symbols

Symbol	Description
*x	Indirection; value pointed to by x
&x	Address of x
!x	Not x
++x	Increment x
−−x	Decrement x
~x	Complement x
−x	Negate x; form two's complement of x
x+y	Add y to x
x−y	Subtract y from x
x*y	Multiply x by y
x/y	Divide y into x
x%y	Modulo x and y (remainder of x/y)
x&y	Bitwise AND x and y
x|y	Bitwise inclusive OR x and y
x^y	Bitwise exclusive OR (XOR) x and y
x<<y	Shift x to the left y bits
x>>y	Shift x to the right y bits
x<y	x less than y
x>y	x greater than y
x==y	Equality; x equal to y
x!=y	x not equal to y
←	Assigns the value on the right to the location identified on the left (same as the C language assignment operator '=')
AP	Argument pointer; register 10 (r10)
count	Count operand
dst	Destination operand
FP	Frame pointer; register 9 (r9)
PC	Program counter; register 15 (r15)
PSW	Processor status word; register 11 (r11)
SEXT(x)	Function that returns x, sign extended through 32 bits.
SP	Stack pointer; register 12 (r12)
*(−−SP)	A pop from the stack; decrement SP by 4 before removing data ( ) from the stack
*(SP++)	A push onto the stack; store data and increment SP by 4
src	Source operand
0xn	Hexadecimal value where n is the digits 0 through 9 and a through f (or A through F); may also be written 0Xn
/*comment*/	A comment, not an operation
{operation}	An operation other than an instruction

3.7.2 Instruction Set Descriptions

The instruction set is described in detail on the following pages.

ADDB2
ADDH2
ADDW2

ADD

Assembler Syntax

ADDB2 *src,dst* Add byte
ADDH2 *src,dst* Add halfword
ADDW2 *src,dst* Add word

Opcodes

0x9F ADDB2
0x9E ADDH2
0x9C ADDW2

Operation

dst ← *dst* + *src*

Address Modes

src all modes

dst all modes except literal or immediate

Condition Flags

N ← 1, if (*dst* + *src*) < 0

Z ← 1, if (*dst* + *src*) == 0

C ← 1, if carry out of sign bit of *dst*

V ← 1, if overflow

Exceptions

Illegal operand exception occurs if literal or immediate mode is used for *dst*.

Integer overflow exception occurs if there is truncation.

Examples

```
ADDB2 $0x100,%r0
ADDH2 %r0,%r3
ADDW2 4(%r3),*$0x110
```

ADDB3
ADDH3
ADDW3

ADD, 3 Address

Assembler Syntax

ADDB3 *src1,src2,dst*	Add byte, 3 address
ADDH3 *src1,src2,dst*	Add halfword, 3 address
ADDW3 *src1,src2,dst*	Add word, 3 address

Opcodes

0xDF ADDB3
0xDE ADDH3
0xDC ADDW3

Operation

dst ← *src1* + *src2*

Address Modes

src1 all modes

src2 all modes

dst all modes except literal or immediate

Condition Flags

N ← 1, if (*src1* + *src2*) < 0

Z ← 1, if (*src1* + *src2*) == 0

C ← 1, if carry out of sign bit of *dst*

V ← 1, if overflow

Exceptions

Illegal operand exception occurs if literal or immediate mode is used for *dst*.

Integer overflow exception occurs if there is truncation.

Examples

```
ADDB3 %r0,%r3,%r5
ADDH3 4(%r2),*$0x110,%r3
ADDW3 *$0x1F0,4(%r1),%r0
```

ARITHMETIC LEFT SHIFT

Assembler Syntax ALSW3 *count,src,dst* Arithmetic left shift word

Opcode 0xC0 ALSW3

Operation *dst* ← *src* << (count & 0x1F) bits

Address Modes

count all modes

src all modes

dst all modes except literal or immediate

Condition Flags

N ← 1, if *dst* < 0

Z ← 1, if *dst* == 0

C ← 0

V ← 0 (see Note)

Exceptions Illegal operand exception occurs if literal or immediate mode is used for *dst*.

Examples

Before: r0

8F	0F	DF	FD

←increasing bits

```
ALSW3 &2,%r0,%r0
```

After: r0

3C	3F	7F	F4

Note All operands are of type word. However, only the five low-order bits of *count* are used; the upper bits are ignored. No bits are shifted past the sign bit, so integer overflow cannot occur. However, the V bit can be set if an expanded-operand type mode changes the type of *dst*. Zeros replace bits that are shifted out. The sign bit is not changed.

ANDB2
ANDH2
ANDW2

ANDB2
ANDH2
ANDW2

AND

Assembler Syntax

ANDB2 *src,dst* AND byte
ANDH2 *src,dst* AND halfword
ANDW2 *src,dst* AND word

Opcodes

0xBB ANDB2
0xBA ANDH2
0xB8 ANDW2

Operation

dst ← *dst* & *src*

Address Modes

src all modes

dst all modes except literal or immediate

Condition Flags

N ← MSB of *dst*

Z ← 1, if *dst* == 0

C ← 0

V ← 1, if result must be truncated to fit *dst* size

Exceptions

Illegal operand exception occurs if literal or immediate mode is used for *dst*.

Examples

```
ANDB2 &7,6(%r1)
ANDH2 %r0,*$result
ANDW2 (%r1),%r4
```

ANDB3
ANDH3
ANDW3

AND, 3 ADDRESS

Assembler Syntax

ANDB3 *src1,src2,dst* AND byte, 3 address
ANDH3 *src1,src2,dst* AND halfword, 3 address
ANDW3 *src1,src2,dst* AND word, 3 address

Opcodes

0xFB ANDB3
0xFA ANDH3
0xF8 ANDW3

Operation

dst ← *src2* + *src1*

Address Modes

src1 all modes

src2 all modes

dst all modes except literal or immediate

Condition Flags

N ← MSB of *dst*

Z ← 1, if *dst* == 0

C ← 0

V ← 1, if result must be truncated to fit *dst* size

Exceptions

Illegal operand exception occurs if literal or immediate mode is used for *dst*.

Examples

```
ANDB3 &0x27,*$0x300,%r6
ANDH3 0x31(%r5),%r0,%r1
ANDW3 %r2,%r1,%r0
```

ARSB3
ARSH3
ARSW3

ARITHMETIC RIGHT SHIFT

Assembler Syntax

ARSB3 *count,src,dst* Arithmetic right shift byte
ARSH3 *count,src,dst* Arithmetic right shift halfword
ARSW3 *count,src,dst* Arithmetic right shift word

Opcodes

0xC7 ARSB3
0xC6 ARSH3
0xC4 ARSW3

Operation

dst ← *src1* >> (*count* & 0x1f) bits

Address Modes

count all modes

src all modes

dst all modes except literal or immediate

Condition Flags

N ← 1, if *dst* < 0

Z ← 1, if *dst* == 0

C ← 0

V ← 0

Exceptions

Illegal operand exception occurs if literal or immediate mode is used for *dst*.

Examples

Before: r0

0F	0F	77	AF

←increasing bits

```
ARSH3 &2,%r0,%r0
```

After: r0

00	00	1D	EB

Note

All operands are of type word. However, only the five low-order bits of *count* are used; the upper bits are ignored. The sign bit (MSB) of *src* is copied as bits are shifted out. The type of *src* does not affect sign extension.

BRANCH ON CARRY CLEAR

Assembler Syntax

BCCB *disp8* Branch on carry clear, byte displacement
BCCH *disp16* Branch on carry clear, halfword displacement

Opcodes

0x53 BCCB
0x52 BCCH

Operation

if (C == 0)
PC ← PC + SEXT(*disp*)

Address Modes

None valid
disp8 = signed 8-bit value

disp16 = signed 16-bit value

Condition Flags

Unchanged

Exceptions

None

Examples

BCCB 0x9
BCCH 0xFF23

BRANCH ON CARRY SET

Assembler Syntax

BCSB *disp8* Branch on carry set, byte displacement
BCSH *disp16* Branch on carry set, halfword displacement

Opcodes

0x5B BCSB
0x5A BCSH

Operation

if (C ==1)
PC ← PC + SEXT(*disp*)

Address Modes

None valid
disp8 = signed 8-bit value

disp16 = signed 16-bit value

Condition Flags

Unchanged

Exceptions

None

Examples

BCSB 0xFF
BCSH 0x1234

BRANCH ON EQUAL

Assembler Syntax

BEB *disp8* Branch on equal, byte displacement
BEH *disp16* Branch on equal, byte displacement

Opcodes

0x7F BEB
0x6F BEB
0x7E BEH
0x6E BEH

Operation

if (Z == 1)
PC ← PC + SEXT(*disp*)

Address Modes

None valid
disp8 = signed 8-bit value

disp16 = signed 16-bit value

Condition Flags

Unchanged

Exceptions

None

Examples

BEB 0xF1
BEH 0x4221

BRANCH ON GREATER THAN (SIGNED)

Assembler Syntax

BGB *disp8*	Branch on greater than, byte displacement (signed)
BGH *disp16*	Branch on greater than, halfword displacement (signed)

Opcodes

0x47 BGB
0x46 BGH

Operation

```
if ((N|Z) == 0)
    PC ← PC + SEXT(disp)
```

Address Modes

None valid

disp8 = signed 8-bit value

disp16 = signed 16-bit value

Condition Flags

Unchanged

Exceptions

None

Examples

```
BGB more
BGH less
```

BRANCH ON GREATER THAN OR EQUAL (SIGNED)

Assembler Syntax

BGEB *disp8* Branch on greater than or equal, byte displacement (signed)
BGEH *disp16* Branch on greater than or equal, halfword displacement (signed)

Opcodes

0x43 BGEB
0x42 BGEH

Operation

```
if ((N == 0)|(Z == 1))
    PC ← PC + SEXT(disp)
```

Address Modes

None valid

disp8 = signed 8-bit value

disp16 = signed 16-bit value

Condition Flags

Unchanged

Exceptions

None

Examples

```
BGEB again
BGEH 0xF102
```

BGEUB
BGEUH

BRANCH ON GREATER THAN OR EQUAL (UNSIGNED)

Assembler Syntax

BGEUB *disp8*	Branch on greater than or equal, byte displacement (unsigned)
BGEUH *disp16*	Branch on greater than or equal, halfword displacement (unsigned)

Opcodes

0x53 BGEUB
0x52 BGEUH

Operation

```
if (C == 0)
    PC ← PC + SEXT(disp)
```

Address Modes

None valid

disp8 = signed 8-bit value

disp16 = signed 16-bit value

Condition Flags

Unchanged

Exceptions

None

Examples

```
BGEUB 0xA1
BGEUH ahead
```

BRANCH ON GREATER THAN (UNSIGNED)

Assembler Syntax

BGUB *disp8*	Branch on greater than, byte displacement (unsigned)
BGUH *disp16*	Branch on greater than, halfword displacement (unsigned)

Opcodes

0x57 BGUB
0x56 BGUH

Operation

```
if ((C|Z) == 0)
    PC ← PC + SEXT(disp)
```

Address Modes

None valid

disp8 = signed 8-bit value

disp16 = signed 16-bit value

Condition Flags

Unchanged

Exceptions

None

Examples

```
BGUB 0xDE
BGUH 0xF123
```

BITB
BITH
BITW

BIT TEST

Assembler Syntax

BITB *src1,src2* Bit test byte
BITH *src1,src2* Bit test halfword
BITW *src1,src2* Bit test word

Opcodes

0x3B BITB
0x3A BITH
0x38 BITW

Operation

temp ← *src2* & *src1*

Address Modes

src1 all modes

src2 all modes

Condition Flags

N ← MSB of temp

Z ← 1, if temp == 0

C ← 0

V ← 0

Exceptions

None

Examples

```
BITB %r0,{uhalf}%r1
BITH *$0xFF,%r3
BITW bit (%r3),(%r0)
```

Note

The final value of temp, a temporary register, determines the setting of the condition codes. Temp is discarded upon completion of the instruction.

BRANCH ON LESS THAN (SIGNED)

Assembler Syntax

BLB *disp8*	Branch on less than, byte displacement (signed)
BLH *disp16*	Branch on less than, halfword displacement (signed)

Opcodes

0x4B BLB
0x4A BLH

Operation

if ((N == 1) & (Z == 0))
PC ← PC + SEXT(*disp*)

Address Modes

None valid

disp8 = signed 8-bit value

disp16 = signed 16-bit value

Condition Flags

Unchanged

Exceptions

None

Examples

```
BLB 0x1F
BLH back
```

BRANCH ON LESS THAN OR EQUAL (SIGNED)

Assembler Syntax

BLEB *disp8*	Branch on less than or equal, byte displacement (signed)
BLEH *disp16*	Branch on less than or equal, halfword displacement (signed)

Opcodes

0x4F BLEB
0x4E BLEH

Operation

if ((N|Z) == 1)
PC ← PC + SEXT(*disp*)

Address Modes

None valid
disp8 = signed 8-bit value

disp16 = signed 16-bit value

Condition Flags

Unchanged

Exceptions

None

Examples

```
BLEB 0x6
BLEH 0xFFF
```

BRANCH ON LESS THAN OR EQUAL (UNSIGNED)

Assembler Syntax

BLEUB *disp8*	Branch on less than or equal, byte displacement (unsigned)
BLEUH *disp16*	Branch on less than or equal, halfword displacement (unsigned)

Opcodes

0x5F BLEUB
0x5E BLEUH

Operation

```
if ((C|Z) == 1)
    PC ← PC + SEXT(disp)
```

Address Modes

None valid

disp8 = signed 8-bit value

disp16 = signed 16-bit value

Condition Flags

Unchanged

Exceptions

None

Examples

```
BLEUB 0x14
BLEUH back
```

BRANCH ON LESS THAN (UNSIGNED)

Assembler Syntax

BLUB *disp8*	Branch on less than byte displacement (unsigned)
BLUH *disp16*	Branch on less than halfword displacement (unsigned)

Opcodes

0x5B BLUB
0x5A BLUH

Operation

```
if (C == 1)
    PC ← PC + SEXT(disp)
```

Address Modes

None valid

disp8 = signed 8-bit value

disp16 = signed 16-bit value

Condition Flags

Unchanged

Exceptions

None

Examples

```
BLUB 0x12
BLUH 0xFF12
```

BRANCH ON NOT EQUAL

Assembler Syntax

BNEB *disp8* Branch on less than, byte displacement
BNEH *disp16* Branch on less than, halfword displacement

Opcodes

0x77 BNEB
0x67 BNEB
0x76 BNEH
0x66 BNEH

Operation

if (Z == 0)
PC ← PC + SEXT(*disp*)

Address Modes

None valid

disp8 = signed 8-bit value

disp16 = signed 16-bit value

Condition Flags

Unchanged

Exceptions

None

Examples

BNEB 0xFE
BNEH 0xFF13

BREAKPOINT TRAP

Assembler Syntax BPT Breakpoint trap

Opcodes 0x2E BPT

Operation /*BPT executes the following processor operation*/
{breakpoint trap}

Address Modes None

Condition Flags Unchanged

Exceptions Generates breakpoint trap exception.

Examples BPT

BRANCH

Assembler Syntax

BRB *disp8* Branch with byte displacement
BRH *disp16* Branch with halfword displacement

Opcodes

0x7B BRB
0x7A BRH

Operation

PC ← PC + SEXT(*disp*)

Address Modes

None valid

disp8 = signed 8-bit value

disp16 = signed 16-bit value

Condition Flags

Unchanged

Exceptions

None

Examples

```
BRB 0xA
BRH 0xFAA
```

BRANCH TO SUBROUTINE

Assembler Syntax

BSBB *disp8* Branch to subroutine, byte displacement
BSBH *disp16* Branch to subroutine, halfword displacement

Opcodes

0x37 BSBB
0x36 BSBH

Operation

*(SP++) ← address of next instruction
PC ← PC + SEXT(*disp*)

Address Modes

None valid

disp8 = signed 8-bit value

disp16 = signed 16-bit value

Condition Flags

Unchanged

Exceptions

None

Examples

```
BSBB sub2
BSBH sub1
```

BVCB
BVCH

BRANCH ON OVERFLOW CLEAR

Assembler Syntax

BVCB *disp8* Branch to subroutine, byte displacement
BVCH *disp16* Branch to subroutine, halfword displacement

Opcodes

0x63 BVCB
0x62 BVCH

Operation

```
if (V == 0)
    PC ← PC + SEXT(disp)
```

Address Modes

None valid

disp8 = signed 8-bit value

disp16 = signed 16-bit value

Condition Flags

Unchanged

Exceptions

None

Examples

```
BVCB 0x7E
BVCH 0x8F21
```

BRANCH ON OVERFLOW SET

Assembler Syntax
BVSB *disp8* Branch on overflow set, byte displacement
BVSH *disp16* Branch on overflow set, halfword displacement

Opcodes
0x6B BVSB
0x6A BVSH

Operation
if (V == 1)
PC ← PC + SEXT(*disp*)

Address Modes
None valid
disp8 = signed 8-bit value

disp16 = signed 16-bit value

Condition Flags
Unchanged

Exceptions
None

Examples
BVS 0xF1
BVSB 0xFF77

CALL PROCEDURE

Assembler Syntax CALL *src,dst* Call procedure

Opcode 0x2C CALL

Operation

tempa ← *&src*
tempb ← *&dst*
*(SP+4) ← AP
*SP ← address of next instruction
SP ← SP+8
PC ← tempb
AP ← tempa

Address Modes

src all modes except literal, register, or immediate

dst all modes except literal, register, or immediate

Condition Flags Unchanged

Exceptions Illegal operand exception occurs if literal, register, expanded-operand type, or immediate mode is used for *src* or *dst*.

Examples CALL −(3*4)(%sp),func1 (see Figure 3-9)

Note Both operands are effective addresses. Temp is a temporary register. CALL sets up the protocol for a C language function call. (Also see Return from procedure.) CALL sets AP to first of the word arguments that the calling function pushed on the stack before executing the call.

CACHE FLUSH

Assembler Syntax CFLUSH Cache flush

Opcode 0x27 CFLUSH

Operation /*CFLUSH executes the following processor operation*/
{all entries in instruction cache are marked invalid}

Address Modes None

Condition Flags Unchanged

Exceptions None

Examples CFLUSH

Notes CFLUSH is a nonprivileged instruction.

This instruction operates identically whether the instruction cache is enabled (PSW<CD>==0) or disabled (PWS<CD>==1).

CLRB
CLRH
CLRW

CLEAR

Assembler Syntax

CLRB *dst* Clear byte
CLRH *dst* Clear halfword
CLRW *dst* Clear word

Opcodes

0x83 CLRB
0x82 CLRH
0x80 CLRW

Operation

dst ← 0

Address Modes

dst all modes except literal or immediate

Condition Flags

N ← 0

Z ← 1

C ← 0

V ← 0

Exceptions

Illegal operand exception occurs if literal or immediate mode is used for *dst*.

Examples

```
CLRB *&0x300
CLRH %r1
CLRW (%r0)
```

CMPB
CMPH
CMPW

CMPB
CMPH
CMPW

COMPARE

Assembler Syntax

CMPB *src1,src2*	Compare byte
CMPH *src1,src2*	Compare halfword
CMPW *src1,src2*	Compare word

Opcodes

0x3F CMPB
0x3E CMPH
0x3C CMPW

Operation

src2 ← *src1*

Address Modes

src1 all modes

src2 all modes

Condition Flags

N ← 1, if *src2* < *src1* (signed)

Z ← 1, if *src2* == *src1*

C ← 1, if *src2* < *src1* (unsigned)

V ← 0

Exceptions

None

Examples

```
CMPB &10,%r0
CMPH (%r0),(%r1)
CMPW *$0x12F7,%r2
```

Note

This instruction sets the condition flags N, Z, and C as if a subtract had been executed. Neither operand is altered. (Also see Test.)

DECREMENT

Assembler Syntax

DECB *dst* Decrement byte
DECH *dst* Decrement halfword
DECW *dst* Decrement word

Opcodes

0x97 DECB
0x96 DECH
0x94 DECW

Operation

dst ← *dst* − 1

Address Modes

dst all modes except literal or immediate

Condition Flags

N ← 1, if (*dst* − 1) < 0

Z ← 1, if (*dst* − 1) == 0

C ← 1, if borrow into sign bit of *dst*

V ← 1, if overflow

Exceptions

Illegal operand exception occurs if literal or immediate mode is used for *dst*.

Integer overflow exception occurs if there is truncation.

Examples

```
DECB 4(%fp)
DECH $result
DECW *$last
```

DIVB2
DIVH2
DIVW2

DIVIDE

Assembler Syntax

DIVB2 *src,dst* Divide byte
DIVH2 *src,dst* Divide halfword
DIVW2 *src,dst* Divide word

Opcodes

0xAF DIVB2
0xAE DIVH2
0xAC DIVW2

Operation

dst ← *dst* / *src*

Address Modes

src all modes

dst all modes except literal or immediate

Condition Flags

N ← 1, if (*dst* / *src*) < 0

Z ← 1, if (*dst* / *src*) == 0

C ← 0

V ← 1, if overflow

Exceptions

Illegal operand exception occurs if literal or immediate mode is used for *dst*.

Integer zero-divide exception occurs if *src* is equal to 0.

Integer overflow exception occurs if there is truncation.

Examples

```
DIVB2 &40,%r6
DIVH2 4(%r3),(%r4)
DIVW2 $first,$last
```

DIVIDE, 3 ADDRESS

Assembler Syntax

DIVB3 *src1,src2,dst* Divide byte, 3 address
DIVH3 *src1,src2,dst* Divide halfword, 3 address
DIVW3 *src1,src2,dst* Divide word, 3 address

Opcodes

0xEF DIVB3
0xEE DIVH3
0xEC DIVW3

Operation

dst ← *src2* / *src1*

Address Modes

src1 all modes

src2 all modes

dst all modes except literal or immediate

Condition Flags

N ← 1, if (*src2* / *src1*) < 0

Z ← 1, if (*src2* / *src1*) == 0

C ← 0

V ← 1, if overflow

Exceptions

Illegal operand exception occurs if literal or immediate mode is used for *dst*.

Integer zero-divide exception occurs if *src1* is equal to 0.

Integer overflow exception occurs if there is truncation.

Examples

```
DIVB3 &0x30,%r3,12(%ap)
DIVH3 &0x3030,(%r2),5(%r2)
DIVW3 &0x304050,(%r1),4(%r1)
```

EXTFB
EXTFH
EXTFW

EXTRACT FIELD

Assembler Syntax

EXTFB	*width,offset,src,dst*	Extract field from byte
EXTFH	*width,offset,src,dst*	Extract field from halfword
EXTFW	*width,offset,src,dst*	Extract field from word

Opcodes

0xCF EXTFB
0xCE EXTFH
0xCC EXTFW

Operation

dst ← FIELD(*offset,width,src*)

Address Modes

width	all modes
offset	all modes
src	all modes
dst	all modes except literal or immediate

Condition Flags

N ← high-order bit of *dst*
Z ← 1, if *dst* == 0
C ← 0
V ← 0 (see Note)

Exceptions

Illegal operand exception occurs if literal or immediate mode is used for *dst*.

Examples

Before: Location L1 = 0x01234567

```
EXTFW &10,&4,L1,%r0
```

After: r0

00	00	04	56

← increasing bits

The field extracted starts at bit 4 of location L1, skipping bits 0 through 3, and extends through bit 14 of L1. These eleven bits are written to bits 0 through 10 of r0; zeros fill the remaining bits of r0.

Note

Only the low-order five bits of *width* and *offset* are examined. If the sum *width* plus *offset* is greater than 32 (bits), then the field wraps around through bit 0 of the base word. The field specified by *width*, *offset*, and *src* is stored, right adjusted, in *dst*. The remaining bits of *dst* are set to 0. If the field is too large for the size of *dst*, the excess high-order bits are discarded and the V flag is set.

EXTENDED OPCODE

Assembler Syntax EXTOP *byte* Extended opcode

Opcode 0x14 EXTOP

Operation /*EXTOP executes the following processor operation*/
{reserved-opcode exception}

Address Modes None valid
byte = 8-bit value

Condition Flags Unchanged

Exceptions Generates reserved opcode exception. See Note.

Examples EXTOP 0x2F

Note The EXTOP opcode is an escape to form additional instructions. The processor does not access *byte* when executing this instruction. Instead, it generates a reserved-opcode exception after decoding the opcode. The operating system's exception handler should access *byte*.

INCB
INCH
INCW

INCREMENT

Assembler Syntax

INCB *dst* Increment byte
INCH *dst* Increment halfword
INCW *dst* Increment word

Opcodes

0x93 INCB
0x92 INCH
0x90 INCW

Operation

dst ← *dst* + 1

Address Modes

dst all modes except literal or immediate

Condition Flags

N ← 1, if (*dst* + 1) < 0

Z ← 1, if (*dst* + 1) == 0

C ← 1, if carry into sign bit of *dst*

V ← 1, if overflow

Exceptions

Illegal operand exception occurs if literal or immediate mode is used for *dst*.

Integer overflow exception occurs if truncation takes place.

Examples

```
INCB 4(%r2)
INCH %r0
INCW (%r1)
```

INSFB
INSFH
INSFW

INSERT FIELD

Assembler Syntax

INSFB	*width,offset,src,dst*	Insert field from byte
INSFH	*width,offset,src,dst*	Insert field from halfword
INSFW	*width,offset,src,dst*	Insert field from word

Opcodes

0xCB	INSFB
0xCA	INSFH
0xC8	INSFW

Operation

FIELD(*offset,width,dst*) ← *src*

Address Modes

width	all modes
offset	all modes
src	all modes
dst	all modes except literal or immediate

Condition Flags

N ← bit 31 of *dst*

Z ← 1, if *dst* == 0

C ← 0

V ← 0 (see Note)

Exceptions

Illegal operand exception occurs if literal or immediate mode is used for *dst*.

Examples

Before: r0

AB	CD	EF	01

r1

00	00	05	67

← increasing bits

```
INSFW &11,&8,%r1,%r0
```

After: r0

AB	C5	67	01

The field insertion starts at bit 8 of r0, skipping bits 0 through 7, and extends through bit 19. Therefore, bits 8 through 19 of r0 now contain the same value as bits 0 through 11 of r1.

Note

Only the low-order five bits of *width* and *offset* are examined. If the sum *width* plus *offset* is greater than 32 (bits), the field wraps around to bit 0 of the destination. Starting with bit 0 of *src*, (*width*+1) bits are placed into *dst* beginning at the bit designated by *offset*. If *dst* is a byte or halfword and (*width*+*offset*) specifies a field that extends beyond *dst*, no bits beyond *dst* are altered but the V flag is set.

JUMP

Assembler Syntax JMP *dst* Jump

Opcode 0x24 JMP

Operation PC ← *&dst*

Address Modes *dst* all modes except literal, register, or immediate

Condition Flags Unchanged

Exceptions Illegal operand exception occurs if literal or immediate mode is used for *dst*.

Examples

```
JMP .L12
```

Note The operand *dst* is an effective address; i.e., the 32-bit address of *dst* is used as the destination rather than the word stored at that address.

JUMP TO SUBROUTINE

Assembler Syntax JSB *dst* Jump to subroutine

Opcode 0x34 JSB

Operation *(SP++) ← address of next instruction
PC ← *&dst*

Address Modes *dst* all modes except literal, register, or immediate

Condition Flags Unchanged

Exceptions Illegal operand exception occurs if literal, expanded-operand type, or immediate mode is used for *dst*.

Examples

```
JSB error
```

Note The operand *dst* is an effective address; i.e., the 32-bit address of *dst* is used as the destination rather than the word at that address.

LLSB3
LLSH3
LLSW3

LOGICAL LEFT SHIFT

Assembler Syntax

LLSB3 *count,src,dst* Logical left shift byte
LLSH3 *count,src,dst* Logical left shift halfword
LLSW3 *count,src,dst* Logical left shift word

Opcodes

0xD3 LLSB3
0xD2 LLSH3
0xD0 LLSW3

Operation

dst ← *src* << (*count* & 0x1F) bits

Address Modes

count all modes

src all modes

dst all modes except literal or immediate

Condition Flags

N ← MSB of *dst*

Z ← 1, if *dst* == 0

C ← 0

V ← 0, if result must be truncated to fit *dst* size

Exceptions

Illegal operand exception occurs if literal or immediate mode is used for *dst*.

Examples

Before: r0 | 0F | 0F | DF | FD |

← increasing bits

```
LLSH3 &2,%r0,%r0
```

After: r0 | FF | FF | 7F | F4 |

Note

Only the five low-order bits of *count* are used; the high-order bits are ignored. Zeros replace the bits shifted out of the low-order bit position (bit 0).

LOGICAL RIGHT SHIFT

Assembler Syntax LRSW3 *count,src,dst* Logical right shift word

Opcode 0xD4 LRSW3

Operation *dst* ← *src* >> (*count* & 0x1F) bits

Address Modes

count all modes

src all modes

dst all modes except literal or immediate

Condition Flags

N ← MSB of *dst*

Z ← 1, if *dst* == 0

C ← 0

V ← 1, if result must be truncated to fit *dst* size

Exceptions Illegal operand exception occurs if literal or immediate mode is used for *dst*.

Examples

Before: r0

C3	C0	00	00

← increasing bits

```
LRSW3 &0x11,%r0,%r0
```

After: r0

00	00	61	E0

Note All operands are type word. However, only the five low-order bits of *count* are used; the high-order bits are ignored. Zeros replace the bits shifted out of the high-order bit position (bit 31).

MCOMB
MCOMH
MCOMW

MOVE COMPLEMENTED

Assembler Syntax

MCOMB *src,dst*	Move complemented byte
MCOMH *src,dst*	Move complemented halfword
MCOMW *src,dst*	Move complemented word

Opcodes

0x8B MCOMB
0x8A MCOMH
0x88 MCOMW

Operation

dst ← ~*src*

Address Modes

src all modes

dst all modes except literal or immediate

Condition Flags

N ← MSB of *dst*

Z ← 1, if *dst* == 0

C ← 0

V ← 1, if result must be truncated to fit *dst* size

Exceptions

Illegal operand exception occurs if literal or immediate mode is used for *dst*.

Examples

Before: r0

12	34	56	78

← increasing bits

MCOMW %r0,%r1

After: r1

ED	CB	A9	87

Note

dst is the one's complement of *src*

MNEGB
MNEGH
MNEGW

MOVE NEGATED

Assembler Syntax

MNEGB *src,dst* Move negated byte
MNEGH *src,dst* Move negated halfword
MNEGW *src,dst* Move negated word

Opcodes

0x8F MNEGB
0x8E MNEGH
0x8C MNEGW

Operation

dst ← −*src*

Address Modes

src all modes

dst all modes except literal or immediate

Condition Flags

N ← MSB of *dst*

Z ← 1, if *dst* == 0

C ← 0

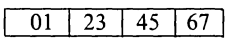
V ← 1, if integer overflow

Exceptions

Illegal operand exception occurs if literal or immediate mode is used for *dst*.

Examples

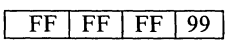
Before: r0



← increasing bits

MNEGB %r0,%r1

After: r1



Note

dst is the two's complement of *src*.

MODB2
MODH2
MODW2

MODULO

Assembler Syntax

MODB2 *src,dst* Modulo byte
MODH2 *src,dst* Modulo halfword
MODW2 *src,dst* Modulo word

Opcodes

0xA7 MODB2
0xA6 MODH2
0xA4 MODW2

Operation

dst ← *dst* % *src*

Address Modes

src all modes

dst all modes except literal or immediate

Condition Flags

N ← 1, if (*dst* % *src*) < 0

Z ← 1, if (*dst* % *src*) == 0

C ← 0

V ← 1, if overflow

Exceptions

Illegal operand exception occurs if literal or immediate mode is used for *dst*.

Integer zero-divide exception occurs if *src* is equal to 0.

Integer overflow exception occurs if there is truncation.

Examples

```
MODB2 &40,%r3
MODH2 4(%r3),%r3
MODW2 %r0,*$result
```

MODB3
MODH3
MODW3

MODULO, 3 ADDRESS

Assembler Syntax

MODB3 *src1,src2,dst*	Modulo byte, 3 address
MODH3 *src1,src2,dst*	Modulo halfword, 3 address
MODW3 *src1,src2,dst*	Modulo word, 3 address

Opcodes

0xE7 MODB3
0xE6 MODH3
0xE4 MODW3

Operation

dst ← *src1* % *src2*

Address Modes

src1 all modes

src2 all modes

dst all modes except literal or immediate

Condition Flags

N ← 1, if (*src1* % *src2*) < 0

Z ← 1, if (*src1* % *src2*) == 0

C ← 0

V ← 1, if overflow

Exceptions

Illegal operand exception occurs if literal or immediate mode is used for *dst*.

Integer zero-divide exception occurs if *src1* is equal to 0.

Integer overflow exception occurs if there is truncation.

Examples

```
MODB3 &40,%r3,0x1101(%r2)
MODH3 %r3,$real,%r3
MODW3 4(%r2),*$0x34,%r0
```

MOVB
MOVH
MOVW

MOVB
MOVH
MOVW

MOVE

Assembler Syntax

MOVB *src*,dst Move byte
MOVH *src*,dst Move halfword
MOVW *src*,dst Move word

Opcodes

0x87 MOVB
0x86 MOVH
0x84 MOVW

Operation

dst ← *src*

Address Modes

src all modes

dst all modes except literal or immediate

Condition Flags

N ← MSB of *dst*

Z ← 1, if *dst* == 0

C ← 0

V ← 1, if result must be truncated to fit *dst* size

See Note

Exceptions

Illegal operand exception occurs if literal or immediate mode is used for *dst*.

Examples

Before:

r0	01	23	45	67
r1	AB	AB	AB	AB

←increasing bits

MOVW %r0,%r1

After:

r0	01	23	45	67
r1	01	23	45	67

NZCV = 0000

MOVB
MOVH
MOVW

Notes

If the expanded-type mode is used for *dst* or for both operands, this instruction can convert data from one type to another. The *src* operand determines the type of extension performed: if *src* is signed byte or halfword, sign extension occurs; if *src* is byte or unsigned halfword, zero extension occurs.

Use the following instructions for conversions if the destination is not a register.

Instruction	Conversion
MOVB {sbyte}*src*,{shalf}*dst*	Signed byte to signed halfword
MOVB {sbyte}*src*,{sword}*dst*	Signed byte to signed word
MOVH *src*,{sword}*dst*	Byte to signed word
MOVB *src*,{shalf}*dst*	Byte to signed halfword
MOVB *src*,{sword}*dst*	Byte to signed word
MOVH {uhalf}*src*,{sword}*dst*	Unsigned halfword to signed word
MOVH *src*,{sbyte}*dst*	Halfword to signed byte
MOVW *src*,{sbyte}*dst*	Word to signed byte
MOVW *src*,{shalf}*dst*	Word to signed halfword

If the destination is a register, use the following instructions for conversions:

Instruction	Conversion
ANDH3 &0xff,*src*,{byte}*dst*	Halfword to byte
ANDW3 &0xff,*src*,{byte}*dst*	Word to byte
MOVW *src,dst*; MOVH *dst,dst*	Word to halfword

The instructions 'MOVW —,%psw' and 'MOVW %psw,—' do not change the condition flags.

MOVE ADDRESS (WORD)

Assembler Syntax MOVAW *src,dst* Move address (word)

Opcode 0x04 MOVAW

Operation *dst* ← *&src*

Address Modes *src* all modes except literal, register, or immediate

dst all modes except literal or immediate

Condition Flags N ← MSB of *dst*

Z ← 1, if *dst* == 0

C ← 0

V ← 0

Exceptions Illegal operand exception occurs if literal, register, or immediate mode is used for *src*, or if literal or immediate mode is used for *dst*.

Examples

Before: r0 | 00 | 00 | 10 | 10 |

r1 | AB | AB | AB | AB |

← increasing bits

MOVAW 4(%r0),%r1

After: r1 | 00 | 00 | 10 | 14 |

Note Source operand type is effective address.

MOVE BLOCK

Assembler Syntax MOVBLW Move block of words

Opcode 0x3019 MOVBLW

Operation

```
while (R2 > 0) {
  *R1 = *R0;
  {disable interrupts}
  -- R2;
  R0=R0+4;
  R1=R1+4;
  {enable interrupts}
}
```

Address Modes None

Condition Flags Unchanged

Exceptions External memory fault may occur in the middle of an iteration.

Examples

Before:

r0	00	00	01	00
r1	00	00	02	00
r2	00	00	00	03

← increasing bits

Assume three word locations starting at 0x100 contain the word values 0x5, 0x10 and 0x20, respectively.

MOVBLW

After:

r0	00	00	01	0C
r1	00	00	02	0C
r2	00	00	00	00

Three word locations starting at 0x200 now also contain 0x5, 0x10 and 0x20, respectively.

Notes

Opcode occupies 16 bits. All operands are implicitly defined in the registers (r0, r1, and r2) and are 32-bit words. These registers must be preset with the following information before executing MOVBLW:

- r0 Address of source
- r1 Address of destination
- r2 Number of words to be moved.

The instruction may be interrupted *only* at the end of an iteration. A memory fault may occur in the middle of an iteration. To restart the instruction after a fault, execute MOVBLW again; the registers are updated after the only memory access that could cause the fault. At each iteration, r0 and r1 are incremented by 4, and r2 is decremented by 1. Execution of MOVBLW is finished when r2 is 0.

MULB2
MULH2
MULW2

MULTIPLY

Assembler Syntax

MULB2 *src,dst* Multiply byte
MULH2 *src,dst* Multiply halfword
MULW2 *src,dst* Multiply word

Opcodes

0xAB MULB2
0xAA MULH2
0xA8 MULW2

Operation

dst ← *dst* * *src*

Address Modes

src all modes

dst all modes except literal or immediate

Condition Flags

N ← 1, if (*dst* * *src*) < 0

Z ← 1, if (*dst* * *src*) == 0

C ← 0

V ← 1, if overflow

Exceptions

Illegal operand exception occurs if literal or immediate mode is used for *dst*.

Integer overflow exception occurs if there is truncation.

Example

MULBH2 %r2,{sbyte}4(%r6)

MULB3
MULH3
MULW3

MULTIPLY, 3 ADDRESS

Assembler Syntax

MULB3 *src1,src2,dst*	Multiply byte, 3 address
MULH3 *src1,src2,dst*	Multiply halfword, 3 address
MULW3 *src1,src2,dst*	Multiply word, 3 address

Opcodes

0xEB MULB3
0xEA MULH3
0xE8 MULW3

Operation

dst ← *src1* * *src2*

Address Modes

src1 all modes

src2 all modes

dst all modes except literal or immediate

Condition Flags

N ← 1, if (*src1* * *src2*) < 0

Z ← 1, if (*src1* * *src2*) == 0

C ← 0

V ← 1, if overflow

Exceptions

Illegal operand exception occurs if literal or immediate mode is used for *dst*.

Integer overflow exception occurs if there is truncation.

Examples

```
MULH3 %r3,*$0x1004,%r4
```

MOVE VERSION NUMBER

Assembler Syntax	MVERNO Move processor version number
Opcode	0x3009 MVERNO
Operation	r0 ← processor version number
Address Modes	None
Condition Flags	Unchanged
Exceptions	None
Example	MVERNO
Note	Opcode occupies 16 bits. Version number is the version of the processor and may range from −128 to +127.

NOP
NOP2
NOP3

NO OPERATION

Assembler Syntax

NOP	No operation, 1 byte
NOP2	No operation, 2 bytes
NOP3	No operation, 3 bytes

Opcodes

0x70	NOP
0x73	NOP2
0x72	NOP3

Operation

None

Address Modes

None

Condition Flags

Unchanged

Exceptions

None

Examples

```
NOP
NOP2
NOP3
```

Notes

The assembler inserts a NOP before instructions (other than branch) that read the PSW. This NOP allows the conditions bits to stabilize. The bytes following NOP2 and NOP3 are generated by the assembler and are ignored by the processor. They may be any value.

ORB2
ORH2
ORW2

OR

Assembler Syntax

ORB2 *src,dst* OR byte
ORH2 *src,dst* OR halfword
ORW2 *src,dst* OR word

Opcodes

0xB3 ORB2
0xB2 ORH2
0xB0 ORW2

Operation

dst ← *dst*|*src*

Address Modes

src all modes

dst all modes except literal or immediate

Condition Flags

N ← MSB of *dst*

Z ← 1, if *dst* == 0

C ← 0

V ← 1, if result must be truncated to fit *dst* size

Exceptions

Illegal operand exception occurs if literal or immediate mode is used for *dst*.

Examples

```
ORB2 &12,4(%fp)
ORH2 %r0,4(%r0)
ORW2 %r3,$result
```

ORB3
ORH3
ORW3

OR, 3 ADDRESS

Assembler Syntax

ORB3 *src1,src2,dst* OR byte, 3 address
ORH3 *src1,src2,dst* OR halfword, 3 address
ORW3 *src1,src2,dst* OR word, 3 address

Opcodes

0xF3 ORB3
0xF2 ORH3
0xF0 ORW3

Operation

dst ← *src2*|*src1*

Address Modes

src1 all modes

src2 all modes

dst all modes except literal or immediate

Condition Flags

N ← MSB of *dst*

Z ← 1, if *dst* == 0

C ← 0

V ← 1, if result must be truncated to fit *dst* size

Exceptions

Illegal operand exception occurs if literal or immediate mode is used for *dst*.

Examples

```
ORB3 &16,*$0x304,%r0
ORH3 %r1,4(%r1),%r1
ORW3 %r2,%r3,%r1
```

POP (WORD)

Assembler Syntax POPW *dst* Pop (word)

Opcode 0x20 POPW

Operation *dst* ← *(−−SP)

Address Modes *dst* all modes except literal or immediate (see Note)

Condition Flags

N ← MSB of *dst*

Z ← 1, if *dst* == 0

C ← 0

V ← 0

Exceptions Illegal operand exception occurs if literal, expanded-operand type, or immediate mode is used for *dst*.

Example

```
POPW (%r2)
```

Note If *dst* is the stack pointer (%sp), the results are indeterminate.

PUSH ADDRESS (WORD)

Assembler Syntax PUSHAW *src* Push address (word)

Opcode 0xE0 PUSHAW

Operation *(SP++) ← *&src*

Address Modes *src* all modes except literal, register, or immediate

Condition Flags

N ← MSB of address of *src*

Z ← 1, if *src* == 0

C ← 0

V ← 0

Exceptions Illegal operand exception occurs if literal, register, expanded-operand type, or immediate mode is used for *src*.

Example

```
PUSHAW 0x14(%r6)
```

Note Source operand type is effective address. This instruction is the same as a move address (MOVAW) instruction, except that the destination for PUSHAW is an implied stack push.

PUSH (WORD)

Assembler Syntax PUSHW *src* Push (word)

Opcode 0xA0 PUSHW

Operation *(SP++) ← *src*

Address Modes *src* all modes

Condition Flags N ← MSB of *src*

Z ← 1, if *src* == 0

C ← 0

V ← 0

Exceptions Illegal operand exception occurs if expanded-operand type addressing mode is used.

Example PUSHW (%r2)

RETURN ON CARRY CLEAR

Assembler Syntax RCC Return on carry clear

Opcode 0x50 RCC

Operation
```
if (C==0)
    PC ← *(--SP)
```

Address Modes None

Condition Flags Unchanged

Exceptions None

Example
```
RCC
```

RETURN ON CARRY SET

Assembler Syntax RCS Return on carry set

Opcode 0x58 RCS

Operation
```
if (C==1)
    PC ← *(--SP)
```

Address Modes None

Condition Flags Unchanged

Exceptions None

Example
```
RCS
```

REQL
REQLU

RETURN ON EQUAL

Assembler Syntax
REQL Return on equal (signed)
REQLU Return on equal (unsigned)

Opcodes
0x7C REQL
0x6C REQLU

Operation
```
if (Z==1)
    PC ← *(--SP)
```

Address Modes
None

Condition Flags
Unchanged

Exceptions
None

Example
REQL

RESTORE REGISTERS

Assembler Syntax RESTORE %r*n* Restore registers

Opcode 0x18 RESTORE

Operation

```
tempa ← FP − 28;
tempb ← *(FP − 28);
tempc ← FP − 24;
while (n != FP){
{
    register[n] ← (tempc)+;
    n+=1;
}
FP ← tempb;
SP ← tempa
```

Address Modes Register mode, where *n* ranges from 0 through 9

Condition Flags Unchanged

Exceptions See Notes.

Example RESTORE %r3

Notes If the operand is not register mode or *n* is not in the range 0 through 9, the results are indeterminate. Although the results are determinate if *n* is 0, 1 or 2, the effect is not that of a register restore in a function-calling sequence.

RESTORE is the inverse of SAVE and should precede a return from procedure (RET). (Also see SAVE and CALL.) The operand %r*n* should be the same as in the corresponding SAVE, where *n* specifies the number of registers (9 − *n*) to be restored for the original function.

RESTORE implements a stack frame for use in the C language function-calling sequence. The instruction can restore up to six registers (from register 8 through register 3) for use by the function. While restoring these registers, it also adjusts SP and FP.

Illegal operand exception occurs if expanded-operand type address mode is used.

RETURN FROM PROCEDURE

Assembler Syntax RET Return from procedure

Opcode 0x18 RET

Operation

```
tempa ← AP;
tempb ← *(SP−4);
tempc ← *(SP−8);
AP    ← tempb;
PC    ← tempc;
SP    ← tempa;
```

Address Modes None

Condition Flags Unchanged

Exceptions None

Example RET

Note The return (RET) is the inverse of the call (CALL) instruction. A restore should precede a return (RET) inside the function being exited. RESTORE sets up the protocol for a C language return from function. RET restores AP, PC, and SP to the values saved on the stack with the corresponding CALL.

RETURN ON GREATER THAN OR EQUAL (SIGNED)

Assembler Syntax RGEQ Return on greater than or equal (signed)

Opcode 0x40 RGEQ

Operation
```
if ((N==0)|(Z==1))
    PC ← *(--SP)
```

Address Modes None

Condition Flags Unchanged

Exceptions None

Example
```
RGEQ
```

RETURN ON GREATER THAN OR EQUAL (UNSIGNED)

Assembler Syntax RGEQU Return on greater than or equal (unsigned)

Opcode 0x50 REGEQU

Operation
```
if (C==0)
    PC ← *(--SP)
```

Address Modes None

Condition Flags Unchanged

Exceptions None

Example RGEQU

RETURN ON GREATER THAN (SIGNED)

Assembler Syntax RGTR Return on greater than (signed)

Opcode 0x44 RGTR

Operation
```
if ((N|Z)==0)
    PC ← *(--SP)
```

Address Modes None

Condition Flags Unchanged

Exceptions None

Example
```
RGTR
```

RETURN ON GREATER THAN (UNSIGNED)

Assembler Syntax RGTRU Return on greater than

Opcode 0x54 RGTRU

Operation
```
if ((C|Z)==0)
    PC ← *(--SP)
```

Address Modes None

Condition Flags Unchanged

Exceptions None

Example
```
RGTRU
```

RETURN ON LESS THAN OR EQUAL (SIGNED)

Assembler Syntax RLEQ Return on less than or equal

Opcode 0x4C RLEQ

Operation
```
if ((N|Z)==1)
    PC ← *(--SP)
```

Address Modes None

Condition Flags Unchanged

Exceptions None

Example RLEQ

RETURN ON LESS THAN OR EQUAL (UNSIGNED)

Assembler Syntax RLEQU Return on less than or equal (unsigned)

Opcode 0x5C RLEQU

Operation

```
if ((C|Z)==1)
    PC ← *(--SP)
```

Address Modes None

Condition Flags Unchanged

Exceptions None

Example RLEQU

RETURN ON LESS THAN (SIGNED)

Assembler Syntax RLSS Return on less than (signed)

Opcode 0x48 RLSS

Operation
```
if ((N==1)&(Z==0))
    PC ← *(--SP)
```

Address Modes None

Condition Flags Unchanged

Exceptions None

Example RLSS

RETURN ON LESS THAN (UNSIGNED)

Assembler Syntax RLSSU Return on less than (unsigned)

Opcode 0x58 RLSSU

Operation
```
if (C==1)
    PC ← *(--SP)
```

Address Modes None

Condition Flags Unchanged

Exceptions None

Example
```
RLSSU
```

RNEQ
RNEQU

RETURN ON NOT EQUAL

Assembler Syntax

RNEQ Return on not equal (signed)
RNEQU Return on not equal (unsigned)

Opcode

0x74 RNEQ
0x64 RNEQU

Operation

```
if (Z==0)
    PC ← *(--SP)
```

Address Modes

None

Condition Flags

Unchanged

Exceptions

None

Example

```
RNEQ
```

ROTATE

Assembler Syntax ROTW *count,src,dst* Rotate word

Opcode 0xD8 ROTW

Operation *dst* ← *src* rotated right (*count* & 0x1F) bits

Address Modes

count all modes

src all modes

dst all modes except literal or immediate

Condition Flags

N ← MSB of *dst*

Z ← 1, if *dst* == 0

C ← 0

V ← 0

Exceptions Illegal operand exception occurs if literal or immediate mode is used for *dst*.

Examples

Before: r0

0F	00	00	7E

← increasing bits

ROTW &0x404,%r0,%r0

After: r0

E0	F0	00	07

Note All operands are type word. However, only the five low-order bits of *count* are used; the high-order bits are ignored.

RETURN FROM SUBROUTINE

Assembler Syntax	RSB Return from subroutine (unconditional)
Opcode	0x78 RSB
Operation	PC ← *(−−SP)
Address Modes	None
Condition Flags	Unchanged
Exceptions	None
Example	RSB

RETURN ON OVERFLOW CLEAR

Assembler Syntax RVC Return on overflow clear

Opcode 0x60 RVC

Operation
```
if (V==0)
    PC ← *(--SP)
```

Address Modes None

Condition Flags Unchanged

Exceptions None

Example RVC

RETURN ON OVERFLOW SET

Assembler Syntax RVS Return on overflow set

Opcode 0x68 RVS

Operation
```
if (V==1)
    PC ← *(--SP)
```

Address Modes None

Condition Flags Unchanged

Exceptions None

Example RVS

SAVE REGISTERS

Assembler Syntax SAVE %r*n* Save registers

Opcode 0x10 SAVE

Operation

```
temp ← SP
*(SP++) ← FP
while (n !=FP){
    *(SP++) ← register[n]
    n+=1;
}
SP ←temp + 28;
FP ← SP;
```

Address Modes Register mode, where *n* ranges from 0 through 9

Condition Flags Unchanged

Exceptions See Notes.

Example SAVE %r3 (see Figure 3-9)

Notes If the operand is not register mode or *n* is not in the range 0 to 9, the results are indeterminate. However, if *n* is 0, 1, or 2, the results are determinate, but SP and FP will not point beyond the register-save area.

Temp is a temporary register, and *n* specifies the number of registers (9 − *n*) to be saved for the calling function.

SAVE implements a stack frame for use in the C language function-calling sequence. It should be the first statement in the called function. (Also see **Restore** and **Return from Procedure** instructions.) SAVE can save up to six registers, from register 8 (r8) through register 3 (r3), freeing them for the new function. After saving these registers, SAVE adjusts SP and FP to point beyond the end of a fixed-size register-save area. Figure 3-9 shows the stack after executing 'SAVE %r3'.

Illegal operand exception occurs if expanded-operand type addressing mode is used.

COPROCESSOR OPERATION (no operands)

Assembler Syntax SPOP *word* Coprocessor operation

Opcode 0x32 SPOP

Operation

```
/* coprocessor operation executes the following
    processor operations */
{ "word" is written out with an access status of
  "coprocessor broadcast" }
{ wait for "coprocessor done" }
{ a word is written into PSW with an access status of
  "coprocessor status fetch" }
```

Address Modes None valid, word = 32-bit value

Condition Flags Unchanged

Exceptions External memory fault may occur.

Example

```
SPOP   0XFFFFFFFF
```

SPOPRS
SPOPRD
SPOPRT

COPROCESSOR OPERATION READ

Assembler Syntax

SPOPRS	*word,src*	Coprocessor operation read single
SPOPRD	*word,src*	Coprocessor operation read double
SPOPPT	*word,src*	Coprocessor operation read triple

Opcode

0x22 SPOPRS
0x02 SPOPRD
0x06 SPOPRT

Operation

```
/* coprocessor operation read executes the following
   processor operations */
{ "word" is written out with an access status of
  "coprocessor broadcast" }
{ "src" is read with an access status of
  "coprocessor data fetch" }
{ wait for "coprocessor done" }
{ a word is written into PSW with an access status of
  "coprocessor status fetch" }
```

Address Modes

word	none valid, 32-bit value
src	all modes except register, literal, or immediate

Condition Flags

Determined by the coprocessor status

Exceptions

External memory fault may occur.

Example

```
SPOPRS  0xF379FFFF,*$0xFF37
SPOPRD  0xFFFFFFFF,%r3
SPOPRT  0x00000000,(%r4)
```

SPOPS2
SPOPD2
SPOPT2

COPROCESSOR OPERATION, 2-ADDRESS

Assembler Syntax

SPOPS2 *word,src,dst*	Coprocessor operation single, 2-address
SPOPD2 *word,src,dst*	Coprocessor operation double, 2-address
SPOPT2 *word,src,dst*	Coprocessor operation triple, 2-address

Opcode

0x23 SPOPWS
0x03 SPOPWD
0x07 SPOPWT

Operation

```
/* coprocessor operation executes the following
     processor operations */
{ "word" is written out with an access status of
  "coprocessor broadcast" }
{ "src" is read with an access status of "coprocessor
  data fetch" }
{ wait for "coprocessor done" }
{ a word is written into PSW with an access status of
  "coprocessor status fetch" }
{ "dst" is written with an access status of
  coprocessor data write" }
```

Address Modes

word	none valid, 32-bit value
src	all modes except register, literal, or immediate
dst	all modes except register, literal, or immediate

Condition Flags

Determined by the coprocessor status

Exceptions

External memory fault may occur.

Example

```
SPOPS2  0xFF,4(%r0)
SPOPD2  0xFFF,%r3
SPOPT2  0xFE,(%r0)
```

COPROCESSOR OPERATION WRITE

Assembler Syntax

```
SPOPWS  word,dst    Coprocessor operation write single
SPOPWD  word,dst    Coprocessor operation write double
SPOPWT  word,dst    Coprocessor operation write triple
```

Opcode

```
0x33  SPOPWS
0x13  SPOPWD
0x17  SPOPWT
```

Operation

```
/* coprocessor operation write executes the following
   processor operations */
{ "word" is written out with an access status of
   "coprocessor broadcast" }
{ wait for "coprocessor done" }
{ a word is written into PSW with an access status of
   coprocessor status fetch" }
{ "dst" is written with an access status of
   coprocessor data write" }
```

Address Modes

word none valid, 32-bit value
dst all modes except register, literal, or immediate

Condition Flags Determined by the coprocessor status.

Exceptions External memory fault may occur.

Example

```
SPOPWS  0x00,%r0
SPOPWD  0x0F,(%r1)
SPOPWT  0x1000,4(%r2)
```

STRING COPY

Assembler Syntax STRCPY String copy

Opcode 0x3035 STRCPY

Operation

```
while ((*r1 = *r0)!=0){
    {disable interrupts}
    r0++;
    r1++;
    {enable interrupts}
}
```

Address Modes None

Condition Flags Unchanged

Exceptions External memory fault may occur in the middle of an iteration.

Examples

Before:

r0	00	00	01	00
r1	00	00	40	00

← increasing bits

The byte locations starting at 0x100 contain the values 0x01, 0x24, 0xE6, 0x7F, 0x11, and 0x00 (location 0x105).

STRCPY

After:

r0	00	00	01	05
r1	00	00	40	05

The byte locations from 0x4000 through 0x4005 now contain the same values as locations 0x100 through 0x105.

Notes

Opcode occupies 16 bits. All operands are defined implicitly in the registers, r0 and r1, that function as byte pointers. These registers must be preset with the following information before executing STRCPY:

r0 Address of source string
r1 Address of destination string

STRCPY implements the string-copy function commonly used in C language. The instruction may be interrupted *only* at the end of an iteration. A memory fault may occur in the middle of an iteration. To restart the instruction after a fault, execute STRCPY again; the registers are updated after the only memory access that could cause the fault. The assignment is a byte move, and both R0 and R1 are incremented by 1 at each iteration. Execution of STRCPY is finished when a null (zero) byte is reached. The null byte is always copied.

STRING END

Assembler Syntax

STREND String end

Opcode

0x301F STREND

Operation

```
while (*r0 !=0){
    r0++;
}
```

Address Modes

None

Condition Flags

Unchanged

Exceptions

External memory fault may occur in the middle of an iteration.

Examples

Before: r0 | 00 | 00 | 04 | 00 |

← increasing bits

The byte locations 0x400 through 0x404 contain the values 0x44, 0x55, 0x01, 0x22, 0x00, respectively.

STREND

After: r0 | 00 | 00 | 04 | 04 |

Notes

Opcode occupies 16 bits. The operand is defined implicitly in the register r0, a byte pointer that must be preset with the starting address of the source C language string. STREND moves the pointer to the end of the string and could be used as part of a string-length or string-concatenation function. The instruction may be interrupted at any time. A memory fault may occur in the middle of an iteration. To restart the instruction after a fault, execute STREND again; the register is updated after the only instruction that could cause the fault. Each iteration tests a byte and increments the pointer r0 by 1. Execution of STREND terminates when a null (zero) byte is found. r0 will be left with the address of the null byte.

SUBB2
SUBH2
SUBW2

SUBTRACT

Assembler Syntax

SUBB2 *src,dst* Subtract byte
SUBH2 *src,dst* Subtract halfword
SUBW2 *src,dst* Subtract word

Opcodes

0xBF SUBB2
0xBE SUBH2
0xBC SUBW2

Operation

dst ← *dst* − *src*

Address Modes

src all modes

dst all modes except literal or immediate

Condition Flags

N ← 1, if (*dst* − *src*) < 0

Z ← 1, if (*dst* − *src*) == 0

C ← 1, if borrow from sign bit of *dst*

V ← 1, if overflow

Exceptions

Illegal operand exception occurs if literal or immediate mode is used for *dst*.

Integer overflow exception occurs if there is truncation.

Examples

```
SUBB2 %r6,*$0x30(%r2)
SUBH2 %r0,$result h
SUBW2 %r3,$resultw
```

SUBB3
SUBH3
SUBW3

SUBTRACT, 3 ADDRESS

Assembler Syntax

SUBB3 *src1,src2,dst* Subtract byte, 3 address
SUBH3 *src1,src2,dst* Subtract halfword, 3 address
SUBW3 *src1,src2,dst* Subtract word, 3 address

Opcodes

0xFF SUBB3
0xFE SUBH3
0xFC SUBW3

Operation

dst ← *src2* − *src1*

Address Modes

src1 all modes

src2 all modes

dst all modes except literal or immediate

Condition Flags

N ← 1, if (*src2* − *src1*) < 0

Z ← 1, if (*src2* − *src1*) == 0

C ← 1, if carry out of sign bit of *dst*

V ← 1, if overflow

Exceptions

Illegal operand exception occurs if literal or immediate mode is used for *dst*.

Integer overflow exception occurs if there is truncation.

Examples

```
SUBB3 %r3,*$0x1005,%r2
SUBH3 %r1,%r3,%r0
SUBW3 $N1,$N2,$result
```

SWAPBI
SWAPHI
SWAPWI

SWAP (INTERLOCKED)

Assembler Syntax

SWAPBI *dst* Swap byte (interlocked)
SWAPHI *dst* Swap halfword (interlocked)
SWAPWI *dst* Swap word (interlocked)

Opcodes

0x1F SWAPBI
0x1E SWAPHI
0x1C SWAPWI

Operation

{set interlock}
tempa ← *dst*
dst ← r0
r0 ← tempa

Address Modes

dst all modes except register, literal, or immediate

Condition Flags

N ← MSB of r0

Z ← 1, if r0 == 0

C ← 0

V ← 0

Exceptions

Illegal operand exception occurs if register, literal, expanded-operand type, or immediate mode is used for *dst*.

Examples

The swap instruction can manipulate interlocks for multiprocessors. Suppose location A is the interlock for a critical section of code, and a nonzero means the lock is busy. Then, the following instructions provide a busy-waiting loop:

```
    MOVW &1,%r0
L1: SWAPWI A
    BNEB L1
```

Note

Final value of r0 sets the condition codes. The SAS code is read interlocked (7) for both the read and write bus transactions.

TSTB
TSTH
TSTW

TEST

Assembler Syntax

TSTB *src* Test byte
TSTH *src* Test halfword
TSTW *src* Test word

Opcodes

0x2B TSTB
0x2A TSTH
0x28 TSTW

Operation

src ← 0

Address Modes

src all modes

Condition Flags

N ← 1, if *src* < 0 (signed)

Z ← 1, if *src* == 0

C ← 0

V ← 0

Exceptions

None

Examples

TSTH 14(%r2)

Note

This instruction only sets condition codes. Its action is the same as a compare instruction, where the first operand is zero, such as

CMPB &0,*src2*

However, test is faster because it is one byte shorter.

XORB2
XORH2
XORW2

EXCLUSIVE OR

Assembler Syntax

XORB2	*src,dst*	Exclusive OR byte
XORH2	*src,dst*	Exclusive OR halfword
XORW2	*src,dst*	Exclusive OR word

Opcodes

0xB7 XORB2
0xB6 XORH2
0xB4 XORW2

Operation

dst ← *dst* ^ *src*

Address Modes

src all modes

dst all modes except literal or immediate

Condition Flags

N ← MSB of *dst*

Z ← 1, if *dst* == 0

C ← 0

V ← 1, if result must be truncated to fit *dst* size

Exceptions

Illegal operand exception occurs if literal or immediate mode is used for *dst*.

Examples

```
XORB2 &40,4(%r4)
XORH2 %r1,$result
XORW2 4(%r1),$result
```

XORB3
XORH3
XORW3

EXCLUSIVE OR, 3 ADDRESS

Assembler Syntax

XORB3 *mask,src,dst*	Exclusive OR byte, 3 address
XORH3 *mask,src,dst*	Exclusive OR halfword, 3 address
XORW3 *mask,src,dst*	Exclusive OR word, 3 address

Opcodes

0xF7 XORB3
0xF6 XORH3
0xF4 XORW3

Operation

dst ← *src* ^*mask*

Address Modes

mask all modes

src all modes

dst all modes except literal or immediate

Condition Flags

N ← MSB of *dst*

Z ← 1, if *dst* == 0

C ← 0

V ← 1, if result must be truncated to fit *dst* size

Exceptions

Illegal operand exception occurs if literal or immediate mode is used for *dst*.

Examples

```
XORB3 &4,*12(%r3),*$0x400
XORH3 %r1,4(%r1),%r0
XORW3 %r0,%r1,%r3
```

3.7.3 Instruction Set Summary by Function

Table 3-12. Data Transfer Instruction Group

Instruction	Mnemonic	Opcode
Move:		
Move byte	MOVB	0x87
Move halfword	MOVH	0x86
Move word	MOVW	0x84
Move address (word)	MOVAW	0x04
Move complemented byte	MCOMB	0x8B
Move complemented halfword	MCOMH	0x8A
Move complemented word	MCOMW	0x88
Move negated byte	MNEGB	0x8F
Move negated halfword	MNEGH	0x8E
Move negated word	MNEGW	0x8C
Move version number	MVERNO	0x3009
Swap (Interlocked):		
Swap byte interlocked	SWAPBI	0x1F
Swap halfword interlocked	SWAPHI	0x1E
Swap word interlocked	SWAPWI	0x1C
Block Operations:		
Move block of words	MOVBLW	0x3019
Field Operations:		
Extract field byte	EXTFB	0xCF
Extract field halfword	EXTFH	0xCE
Extract field word	EXTFW	0xCC
Insert field byte	INSFB	0xCB
Insert field halfword	INSFH	0xCA
Insert field word	INSFW	0xC8
String Operations:		
String copy	STRCPY	0x3035
String end	STREND	0x301F

Table 3-13. Arithmetic Instruction Group

Instruction	Mnemonic	Opcode
Add:		
Add byte	ADDB2	0x9F
Add halfword	ADDH2	0x9E
Add word	ADDW2	0x9C
Add byte, 3-address	ADDB3	0xDF
Add halfword, 3-address	ADDH3	0xDE
Add word, 3-address	ADDW3	0xDC

Table 3-13. Arithmetic Instruction Group (Continued)

Instruction	Mnemonic	Opcode
Subtract:		
Subtract byte	SUBB2	0xBF
Subtract halfword	SUBH2	0xBE
Subtract word	SUBW2	0xBC
Subtract byte, 3-address	SUBB3	0xFF
Subtract halfword, 3-address	SUBH3	0xFE
Subtract word, 3-address	SUBW3	0xFC
Increment:		
Increment byte	INCB	0x93
Increment halfword	INCH	0x92
Increment word	INCW	0x90
Decrement:		
Decrement byte	DECB	0x97
Decrement halfword	DECH	0x96
Decrement word	DECW	0x94
Multiply:		
Multiply byte	MULB2	0xAB
Multiply halfword	MULH2	0xAA
Multiply word	MULW2	0xA8
Multiply byte, 3-address	MULB3	0xEB
Multiply halfword, 3-address	MULH3	0xEA
Multiply word, 3-address	MULW3	0xE8
Divide:		
Divide byte	DIVB2	0xAF
Divide halfword	DIVH2	0xAE
Divide word	DIVW2	0xAC
Divide byte, 3-address	DIVB3	0xEF
Divide halfword, 3-address	DIVH3	0xEE
Divide word, 3-address	DIVW3	0xEC
Modulo:		
Modulo byte	MODB2	0xA7
Modulo halfword	MODH2	0xA6
Modulo word	MODW2	0xA4
Modulo byte, 3-address	MODB3	0xE7
Modulo halfword, 3-address	MODH3	0xE6
Modulo word, 3-address	MODW3	0xE4
Arithmetic Shift:		
Arithmetic left shift word	ALSW3	0xC0
Arithmetic right shift byte	ARSB3	0xC7
Arithmetic right shift halfword	ARSH3	0xC6
Arithmetic right shift word	ARSW3	0xC4

Table 3-14. Logical Group

Instruction	Mnemonic	Opcode
AND:		
AND byte	ANDB2	0xBB
AND halfword	ANDH2	0xBA
AND word	ANDW2	0xB8
AND byte, 3-address	ANDB3	0xFB
AND halfword, 3-address	ANDH3	0xFA
AND word, 3-address	ANDW3	0xF8
Exclusive OR (XOR):		
Exclusive OR byte	XORB2	0xB7
Exclusive OR halfword	XORH2	0xB6
Exclusive OR word	XORW2	0xB4
Exclusive OR byte, 3-address	XORB3	0xF7
Exclusive OR halfword, 3-address	XORH3	0xF6
Exclusive OR word, 3-address	XORW3	0xF4
OR:		
OR byte	ORB2	0xB3
OR halfword	ORH2	0xB2
OR word	ORW2	0xB0
OR byte, 3-address	ORB3	0xF3
OR halfword, 3-address	ORH2	OxF2
OR word, 3-address	ORW3	OxF0
Compare or Test:		
Compare byte	CMPB	0x3F
Compare halfword	CMPH	0x3E
Compare word	CMPW	0x3C
Test byte	TSTB	0x2B
Test halfword	TSTH	0x2A
Test word	TSTW	0x28
Bit test byte	BITB	0x3B
Bit test halfword	BITH	0x3A
Bit test word	BITW	0x38
Clear:		
Clear byte	CLRB	0x83
Clear halfword	CLRH	0x82
Clear word	CLRW	0x80
Rotate or Logical Shift:		
Rotate word	ROTW	0xD8
Logical left shift byte	LLSB3	0xD3
Logical left shift halfword	LLSH3	0xD2
Logical left shift word	LLSW3	0xD0
Logical right shift word	LRSW3	0xD4

Table 3-15. Program Control Instructions		
Instruction	**Mnemonic**	**Opcode**
Unconditional Transfer:		
Branch with byte (8-bit) displacement	BRB	0x7B
Branch with halfword (16-bit) displacement	BRH	0x7A
Jump	JMP	0x24
Conditional Transfers:		
Branch on carry clear byte	BCCB	0x53*
Branch on carry clear halfword	BCCH	0x52*
Branch on carry set byte	BCSB	0x5B
Branch on carry set halfword	BCSH	0x5A*
Branch on overflow clear, byte displacement	BVCB	0x63
Branch on overflow clear, halfword displacement	BVCH	0x62
Branch on overflow set, byte displacement	BVSB	0x6B
Branch on overflow set, halfword displacement	BVSH	0x6A
Branch on equal byte (duplicate)	BEB	0x6F
Branch on equal byte	BEB	0x7F
Branch on equal halfword (duplicate)	BEH	0x6E
Branch on equal halfword	BEH	0x7E
Branch on not equal byte (duplicate)	BNEB	0x67
Branch on not equal byte	BNEB	0x77
Branch on not equal halfword (duplicate)	BNEH	0x66
Branch on not equal halfword	BNEH	0x76
Branch on less than byte (signed)	BLB	0x4B
Branch on less than halfword (signed)	BLH	0x4A
Branch on less than byte (unsigned)	BLUB	0x5B*
Branch on less than halfword (unsigned)	BLUH	0x5A*
Branch on less than or equal byte (signed)	BLEB	0x4F
Branch on less than or equal halfword (signed)	BLEH	0x4E
Branch on less than or equal byte (unsigned)	BLEUB	0x5F
Branch on less than or equal halfword (unsigned)	BLEUH	0x5E
Branch on greater than byte (signed)	BGB	0x47
Branch on greater than halfword (signed)	BGH	0x46
Branch on greater than byte (unsigned)	BGUB	0x57
Branch on greater than halfword (unsigned)	BGUH	0x56
Branch on greater than or equal byte (signed)	BGEB	0x43
Branch on greater than or equal halfword (signed)	BGEH	0x42
Branch on greater than or equal byte (unsigned)	BGEUB	0x53*
Branch on greater than or equal halfword (unsigned)	BGEUH	0x52*

* Indicates that opcode matches another instruction but operation is the same.

Table 3-15. Program Control Instructions (Continued)		
Instruction	**Mnemonic**	**Opcode**
Conditional Transfers (Continued):		
Return on carry clear	RCC	0x50*
Return on carry set	RCS	0x58*
Return on overflow clear	RVC	0x60
Return on overflow set	RVS	0x68
Return on equal (unsigned)	REQLU	0x6C
Return on equal (signed)	REQL	0x7C
Return on not equal (unsigned)	RNEQU	0x64
Return on not equal (signed)	RNEQ	0x74
Return on less than (signed)	RLSS	0x48
Return on less than (unsigned)	RLSSU	0x58*
Return on less than or equal (signed)	RLEQ	0x4C
Return on less than or equal (unsigned)	RLEQU	0x5C
Return on greater than (signed)	RGTR	0x44
Return on greater than (unsigned)	RGTRU	0x54
Return on greater than or equal (signed)	RGEQ	0x40
Return on greater than or equal (unsigned)	RGEQU	0x50*
Subroutine Transfer:		
Branch to subroutine, byte displacement	BSBB	0x37
Branch to subroutine, halfword displacement	BSBH	0x36
Jump to subroutine	JSB	0x34
Return from subroutine	RSB	0x78
Procedure Transfer:		
Save registers	SAVE	0x10
Restore registers	RESTORE	0x18
Call procedure	CALL	0x2C
Return from procedure	RET	0x08

* Indicates that opcode matches another instruction but operation is the same.

Table 3-16. Coprocessor Instructions		
Instruction	**Mnemonic**	**Opcode**
Coprocessor operation	SPOP	0x32
Coprocessor operation read single	SPOPRS	0x22
Coprocessor operation read double	SPOPRD	0x02
Coprocessor operation read triple	SPOPRT	0x06
Coprocessor operation single 2-address	SPOPS2	0x23
Coprocessor operation double 2-address	SPOPD2	0x03
Coprocessor operation triple 2-address	SPOPT2	0x07
Coprocessor operation write single	SPOPWS	0x33
Coprocessor operation write double	SPOPWD	0x13
Coprocessor operation write triple	SPOPWT	0x17

Table 3-17. Stack and Miscellaneous Instructions		
Instruction	**Mnemonic**	**Opcode**
Stack Operations:		
Push address word	PUSHAW	0xE0
Push word	PUSHW	0xA0
Pop word	POPW	0x20
Miscellaneous:		
No operation, 1 byte	NOP	0x70
No operation, 2 bytes	NOP2	0x73
No operation, 3 bytes	NOP3	0x72
Breakpoint trap	BPT	0x2E
Extended opcode	EXTOP	0x14
Cache flush	CFLUSH	0x27

3.7.4 Instruction Set Summary by Mnemonic

Table 3-18. Instruction Set Summary by Mnemonic

Mnemonic	Opcode	Instruction
ADDB2	0x9F	Add byte
ADDB3	0xDF	Add byte, 3-address
ADDH2	0x9E	Add halfword
ADDH3	0xDE	Add halfword, 3-address
ADDW2	0x9C	Add word
ADDW3	0xDC	Add word, 3-address
ALSW3	0xC0	Arithmetic left shift word
ANDB2	0xBB	AND byte
ANDB3	0xFB	AND byte, 3-address
ANDH2	0xBA	AND halfword
ANDH3	0xFA	AND halfword, 3-address
ANDW2	0xB8	AND word
ANDW3	0xF8	AND word, 3-address
ARSB3	0xC7	Arithmetic right shift byte
ARSH3	0xC6	Arithmetic right shift halfword
ARSW3	0xC4	Arithmetic right shift word
BCCB	0x53*	Branch on carry clear byte
BCCH	0x52*	Branch on carry clear halfword
BCSB	0x5B*	Branch on carry set byte
BCSH	0x5A*	Branch on carry set halfword
BEB	0x6F	Branch on equal byte (duplicate)
BEB	0x7F	Branch on equal byte
BEH	0x6E	Branch on equal halfword (duplicate)
BEH	0x7E	Branch on equal halfword
BGB	0x47	Branch on greater than byte (signed)
BGEB	0x43	Branch on greater than or equal byte (signed)
BGEH	0x42	Branch on greater than or equal halfword (signed)
BGEUB	0x53*	Branch on greater than or equal byte (unsigned)
BGEUH	0x52*	Branch on greater than or equal halfword (unsigned)
BGH	0x46	Branch on greater than halfword (signed)
BGUB	0x57	Branch on greater than byte (unsigned)
BGUH	0x56	Branch on greater than halfword (unsigned)
BITB	0x3B	Bit test byte
BITH	0x3A	Bit test halfword
BITW	0x38	Bit test word
BLB	0x4B	Branch on less than byte (signed)
BLEB	0x4F	Branch on less than or equal byte (signed)
BLEH	0x4E	Branch on less than or equal halfword (signed)

* Indicates that opcode matches another instruction but operation is the same.

Table 3-18. Instruction Set Summary by Mnemonic (Continued)		
Mnemonic	**Opcode**	**Instruction**
BLEUB	0x5F	Branch on less than or equal byte (unsigned)
BLEUH	0x5E	Branch on less than or equal halfword (unsigned)
BLH	0x4A	Branch on less than halfword (signed)
BLUB	0x5B*	Branch on less than byte (unsigned)
BLUH	0x5A*	Branch on less than halfword (unsigned)
BNEB	0x67	Branch on not equal byte (duplicate)
BNEB	0x77	Branch on not equal byte
BNEH	0x66	Branch on not equal halfword (duplicate)
BNEH	0x76	Branch on not equal halfword
BPT	0x2E	Breakpoint trap
BRB	0x7B	Branch with byte (8-bit) displacement
BRH	0x7A	Branch with halfword (16-bit) displacement
BSBB	0x37	Branch to subroutine, byte displacement
BSBH	0x36	Branch to subroutine, halfword displacement
BVCB	0x63	Branch on overflow clear, byte displacement
BVCH	0x62	Branch on overflow clear, halfword displacement
BVSB	0x6B	Branch on overflow set, byte displacement
BVSH	0x6A	Branch on overflow set, halfword displacement
CALL	0x2C	Call procedure
CFLUSH	0x27	Cache flush
CLRB	0x83	Clear byte
CLRH	0x82	Clear halfword
CLRW	0x80	Clear word
CMPB	0x3F	Compare byte
CMPH	0x3E	Compare halfword
CMPW	0x3C	Compare word
DECB	0x97	Decrement byte
DECH	0x96	Decrement halfword
DECW	0x94	Decrement word
DIVB2	0xAF	Divide byte
DIVB3	0xEF	Divide byte 3-address
DIVH2	0xAE	Divide halfword
DIVH3	0xEE	Divide halfword, 3-address
DIVW2	0xAC	Divide word
DIVW3	0xEC	Divide word, 3-address
EXTFB	0xCF	Extract field byte
EXTFH	0xCE	Extract field halfword
EXTFW	0xCC	Extract field word
EXTOP	0x14	Extended opcode
INCB	0x93	Increment byte
INCH	0x92	Increment halfword
INCW	0x90	Increment word
INSFB	0xCB	Insert field byte

* Indicates that opcode matches another instruction but operation is the same.

Table 3-18. Instruction Set Summary by Mnemonic (Continued)

Mnemonic	Opcode	Instruction
INSFH	0xCA	Insert field halfword
INSFW	0xC8	Insert field word
JMP	0x24	Jump
JSB	0x34	Jump to subroutine
LLSB3	0xD3	Logical left shift byte
LLSH3	0xD2	Logical left shift halfword
LLSW3	0xD0	Logical left shift word
LRSW3	0xD4	Logical right shift word
MCOMB	0x8B	Move complemented byte
MCOMH	0x8A	Move complemented halfword
MCOMW	0x88	Move complemented word
MNEGB	0x8F	Move negated byte
MNEGH	0x8E	Move negated halfword
MNEGW	0x8C	Move negated word
MODB2	0xA7	Modulo byte
MODB3	0xE7	Modulo byte, 3-address
MODH2	0xA6	Modulo halfword
MODH3	0xE6	Modulo halfword, 3-address
MODW2	0xA4	Modulo word
MODW3	0xE4	Modulo word, 3-address
MOVAW	0x04	Move address (word)
MOVB	0x87	Move byte
MOVBLW	0x3019	Move block of words
MOVH	0x86	Move halfword
MOVW	0x84	Move word
MULB2	0xAB	Multiply byte
MULB3	0xEB	Multiply byte, 3-address
MULH2	0xAA	Multiply halfword
MULH3	0xEA	Multiply halfword, 3-address
MULW2	0xA8	Multiply word
MULW3	0xE8	Multiply word, 3-address
MVERNO	0x3009	Move version number
NOP	0x70	No operation, 1 byte
NOP2	0x73	No operation, 2 bytes
NOP3	0x72	No operation, 3 bytes
ORB2	0xB3	OR byte
ORB3	0xF3	OR byte, 3-address
ORH2	0xB2	OR halfword
ORH3	0xF2	OR halfword, 3-address
ORW2	0xB0	OR word
ORW3	0xF0	OR word, 3-address
POPW	0x20	Pop word
PUSHAW	0xE0	Push address word
PUSHW	0xA0	Push word

Table 3-18. Instruction Set Summary by Mnemonic (Continued)		
Mnemonic	**Opcode**	**Instruction**
RCC	0x50*	Return on carry clear
RCS	0x58*	Return on carry set
REQLU	0x6C	Return on equal (unsigned)
REQL	0x7C	Return on equal (signed)
RESTORE	0x18	Restore registers
RET	0x08	Return from procedure
RGEQ	0x40	Return on greater than or equal (signed)
RGEQU	0x50*	Return on greater than or equal (unsigned)
RGTR	0x44	Return on greater than (signed)
RGTRU	0x54	Return on greater than (unsigned)
RLEQ	0x4C	Return on less than or equal (signed)
RLEQU	0x5C	Return on less than or equal (unsigned)
RLSS	0x48	Return on less than (signed)
RLSSU	0x58*	Return on less than (unsigned)
RNEQU	0x64	Return on not equal (unsigned)
RNEQ	0x74	Return on not equal (signed)
ROTW	0xD8	Rotate word
RSB	0x78	Return from subroutine
RVC	0x60	Return on overflow clear
RVS	0x68	Return on overflow set
SAVE	0x10	Save registers
SPOP	0x32	Coprocessor operation
SPOPRS	0x22	Coprocessor operation read single
SPOPRD	0x02	Coprocessor operation read double
SPOPRT	0x06	Coprocessor operation read triple
SPOPS2	0x23	Coprocessor operation single 2-address
SPOPD2	0x03	Coprocessor operation double 2-address
SPOPT2	0x07	Coprocessor operation triple 2-address
SPOPWS	0x33	Coprocessor operation write single
SPOPWD	0x13	Coprocessor operation write double
SPOPWT	0x17	Coprocessor operation write triple
STRCPY	0x3035	String copy
STREND	0x301F	String end
SUBB2	0xBF	Subtract byte
SUBB3	0xFF	Subtract byte, 3-address
SUBH2	0xBE	Subtract halfword
SUBH3	0xFE	Subtract halfword, 3-address
SUBW2	0xBC	Subtract word
SUBW3	0xFC	Subtract word, 3-address

* Indicates that opcode matches another instruction but operation is the same.

Table 3-18. Instruction Set Summary by Mnemonic (Continued)		
Mnemonic	**Opcode**	**Instruction**
SWAPBI	0x1F	Swap byte interlocked
SWAPHI	0x1E	Swap halfword interlocked
SWAPWI	0x1C	Swap word interlocked
TSTB	0x2B	Test byte
TSTH	0x2A	Test halfword
TSTW	0x28	Test word
XORB2	0xB7	Exclusive OR byte
XORB3	0xF7	Exclusive OR byte, 3-address
XORH2	0xB6	Exclusive OR halfword
XORH3	0xF6	Exclusive OR halfword, 3-address
XORW2	0xB4	Exclusive OR word
XORW3	0xF4	Exclusive OR word, 3-address

3.7.5 Instruction Set Summary by Opcode

Table 3-19. Instruction Set Summary by Opcode		
Menemonic	**Opcode**	**Instruction**
SPOPRD	0x02	Coprocessor operation read double
SPOPD2	0x03	Coprocessor operation double, 2-address
MOVAW	0x04	Move address (word)
SPOPRT	0x06	Coprocessor operation read triple
SPOPT2	0x07	Coprocessor operation triple, 2-address
RET	0x08	Return from procedure
SAVE	0x10	Save registers
SPOPWD	0x13	Coprocessor operation write double
EXTOP	0x14	Extended opcode
SPOPWT	0x17	Coprocessor operation write triple
RESTORE	0x18	Restore registers
SWAPWI	0x1C	Swap word interlocked
SWAPHI	0x1E	Swap halfword interlocked
SWAPBI	0x1F	Swap byte interlocked
POPW	0x20	Pop word
SPOPRS	0x22	Coprocessor operation read single
SPOPS2	0x23	Coprocessor operation single, 2-address
JMP	0x24	Jump
TSTW	0x28	Test word
TSTH	0x2A	Test halfword
TSTB	0x2B	Test byte
CALL	0x2C	Call procedure
BPT	0x2E	Breakpoint trap
MVERNO	0x3009	Move version number
MOVBLW	0x3019	Move block of words
STREND	0x301F	String end
STRCPY	0x3035	String copy

Table 3-19. Instruction Set Summary by Opcode (Continued)

Mnemonic	Opcode	Instruction
SPOP	0x32	Coprocessor operation
SPOPWS	0x33	Coprocessor operation write single
JSB	0x34	Jump to subroutine
BSBH	0x36	Branch to subroutine, halfword displacement
BSBB	0x37	Branch to subroutine, byte displacement
BITW	0x38	Bit test word
BITH	0x3A	Bit test halfword
BITB	0x3B	Bit test byte
CMPW	0x3C	Compare word
CMPH	0x3E	Compare halfword
CMPB	0x3F	Compare byte
RGEQ	0x40	Return on greater than or equal (signed)
BGEH	0x42	Branch on greater than or equal halfword (signed)
BGEB	0x43	Branch on greater than or equal byte (signed)
RGTR	0x44	Return on greater than (signed)
BGH	0x46	Branch on greater than halfword (signed)
BGB	0x47	Branch on greater than byte (signed)
RLSS	0x48	Return on less than (signed)
BLH	0x4A	Branch on less than halfword (signed)
BLB	0x4B	Branch on less than byte (signed)
RLEQ	0x4C	Return on less than or equal (signed)
BLEH	0x4E	Branch on less than or equal halfword (signed)
BLEB	0x4F	Branch on less than or equal byte (signed)
RCC	0x50*	Return on carry clear
RGEQU	0x50*	Return on greater than or equal (unsigned)
BCCH	0x52*	Branch on carry clear halfword
BGEUH	0x52*	Branch on greater than or equal halfword (unsigned)
BCCB	0x53*	Branch on carry clear byte
BGEUB	0x53*	Branch on greater than or equal byte (unsigned)
RGTRU	0x54	Return on greater than (unsigned)
BGUH	0x56	Branch on greater than halfword (unsigned)
BGUB	0x57	Branch on greater than byte (unsigned)
RCS	0x58*	Return on carry set
RLSSU	0x58*	Return on less than (unsigned)
BCSH	0x5A*	Branch on carry set halfword
BLUH	0x5A*	Branch on less than halfword (unsigned)
BCSB	0x5B*	Branch on carry set byte
BLUB	0x5B*	Branch on less than byte (unsigned)
RLEQU	0x5C	Return on less than or equal (unsigned)
BLEUH	0x5E	Branch on less than or equal halfword (unsigned)
BLEUB	0x5F	Branch on less than or equal byte (unsigned)

* Indicates that opcode matches another instruction but operation is the same.

Table 3-19. Instruction Set Summary by Opcode (Continued)

Mnemonic	Opcode	Instruction
RVC	0x60	Return on overflow clear
BVCH	0x62	Branch on overflow clear, halfword displacement
BVCB	0x63	Branch on overflow clear, byte displacement
RNEQU	0x64	Return on not equal (unsigned)
BNEH	0x66	Branch on not equal halfword (duplicate)
BNEB	0x67	Branch on not equal byte (duplicate)
RVS	0x68	Return on overflow set
BVSH	0x6A	Branch on overflow set, halfword displacement
BVSB	0x6B	Branch on overflow set, byte displacement
REQLU	0x6C	Return on equal (unsigned)
BEH	0x6E	Branch on equal halfword (duplicate)
BEB	0x6F	Branch on equal byte (duplicate)
NOP	0x70	No operation, 1 byte
NOP3	0x72	No operation, 3 bytes
NOP2	0x73	No operation, 2 bytes
RNEQ	0x74	Return on not equal (signed)
BNEH	0x76	Branch on not equal halfword
BNEB	0x77	Branch on not equal
RSB	0x78	Return from subroutine
BRH	0x7A	Branch with halfword (16-bit) displacement
BRH	0x7B	Branch with byte (8-bit) displacement
REQL	0x7C	Return on equal (signed)
BEH	0x7E	Branch on equal halfword
BEB	0x7F	Branch on equal byte
CLRW	0x80	Clear word
CLRH	0x82	Clear halfword
CLRB	0x83	Clear byte
MOVW	0x84	Move word
MOVH	0x86	Move halfword
MOVB	0x87	Move byte
MCOMW	0x88	Move complemented word
MCOMH	0x8A	Move complemented halfword
MCOMB	0x8B	Move complemented byte
MNEGW	0x8C	Move negated word
MNEGH	0x8E	Move negated halfword
MNEGB	0x8F	Move negated byte
INCW	0x90	Increment word
INCH	0x92	Increment halfword
INCB	0x93	Increment byte
DECW	0x94	Decrement word
DECH	0x96	Decrement halfword
DECB	0x97	Decrement byte
ADDW2	0x9C	Add word
ADDH2	0x9E	Add halfword
ADDB2	0x9F	Add byte

Table 3-19. Instruction Set Summary by Opcode (Continued)		
Mnemonic	**Opcode**	**Instruction**
PUSHW	0xA0	Push word
MODW2	0xA4	Modulo word
MODH2	0xA6	Modulo halfword
MODB2	0xA7	Modulo byte
MULW2	0xA8	Multiply word
MULH2	0xAA	Multiply halfword
MULB2	0xAB	Multiply byte
DIVW2	0xAC	Divide word
DIVH2	0xAE	Divide halfword
DIVB2	0xAF	Divide byte
ORW2	0xB0	OR word
ORH2	0xB2	OR halfword
ORB2	0xB3	OR byte
XORW2	0xB4	Exclusive OR word
XORH2	0xB6	Exclusive OR halfword
XORB2	0xB7	Exclusive OR byte
ANDW2	0xB8	AND word
ANDH2	0xBA	AND halfword
ANDB2	0xBB	AND byte
SUBW2	0xBC	Subtract word
SUBH2	0xBE	Subtract halfword
SUBB2	0xBF	Subtract byte
ALSW3	0xC0	Arithmetic left shift word
ARSW3	0xC4	Arithmetic right shift word
ARSH3	0xC6	Arithmetic right shift halfword
ARSB3	0xC7	Arithmetic right shift byte
INSFW	0xC8	Insert field word
INSFH	0xCA	Insert field halfword
INSFB	0xCB	Insert field byte
EXTFW	0xCC	Extract field word
EXTFH	0xCE	Extract field halfword
EXTFB	0xCF	Extract field byte
LLSW3	0xD0	Logical left shift word
LLSH3	0xD2	Logical left shift halfword
LLSB3	0xD3	Logical left shift byte
LRSW3	0xD4	Logical right shift word
ROTW	0xD8	Rotate word
ADDW3	0xDC	Add word, 3-address
ADDH3	0xDE	Add halfword, 3-address
ADDB3	0xDF	Add byte, 3-address

Table 3-19. Instruction Set Summary by Opcode (Continued)		
Mnemonic	**Opcode**	**Instruction**
PUSHAW	0xE0	Push address word
MODW3	0xE4	Modulo word, 3-address
MODH3	0xE6	Modulo halfword, 3-address
MODB3	0xE7	Modulo byte, 3-address
MULW3	0xE8	Multiply word, 3-address
MULH3	0xEA	Multiply halfword, 3-address
MULB3	0xEB	Multiply byte, 3-address
DIVW3	0xEC	Divide word, 3-address
DIVH3	0xEE	Divide halfword, 3-address
DIVB3	0xEF	Divide byte, 3-address
ORW3	0xF0	OR word, 3-address
ORH3	0xF2	OR halfword, 3-address
ORB3	0xF3	OR byte, 3-address
XORW3	0xF4	Exclusive OR word, 3-address
XORH3	0xF6	Exclusive OR halfword, 3-address
XORB3	0xF7	Exclusive OR byte, 3-address
ANDW3	0xF8	AND word, 3-address
ANDH3	0xFA	AND halfword, 3-address
ANDB3	0xFB	AND byte, 3-address
SUBW3	0xFC	Subtract word, 3-address
SUBH3	0xFE	Subtract halfword, 3-address
SUBB3	0xFF	Subtract byte, 3-address

Chapter 4

Operating System Considerations

CHAPTER 4. OPERATING SYSTEM CONSIDERATIONS

CONTENTS

4. OPERATING SYSTEM CONSIDERATIONS

The *WE* 32100 Microprocessor allows cost-effective design of operating systems by providing the system designer with special purpose operating system instructions and an architecture that supports process-oriented operating system design. In general, a *process* is a separately scheduled, independently executed unit of activity. It generally consists of routines (functions) that perform a major task (such as a program manager, a file manager, or a memory manager). To make full use of the power of the *WE* 32100 Microprocessor as an execution vehicle for today's efficient process-oriented operating systems, this chapter presents the operating system considerations important to the system designer.

The typical operating system for the *WE* 32100 Microprocessor schedules and initiates all processes, handles error conditions (*exceptions* to normal processing), provides system security, and resets the microprocessor when appropriate. Processes are scheduled through common scheduling algorithms and are initiated through a *process switch*. A process switch is an explicit or implicit request that changes the process controlling the microprocessor. An explicit process switch is invoked by execution of one of the special operating system instructions. An implicit process switch occurs as a result of a reset request, some interrupt requests, or certain exception conditions. In theory, the microprocessor can handle an unlimited number of processes, but real limits are imposed by the operating system design (i.e., limiting the size of the interrupt stack). System security is enforced by the microprocessor and by the *WE* 32101 Memory Management Unit (MMU), an integral part of a virtual memory-based operating system using the *WE* 32100 Microprocessor. The microprocessor is reset by the operating system through a reset exception handler process. This handler should initialize the system hardware and reload the operating system.

4.1 FEATURES OF THE OPERATING SYSTEM

As part of its architecture the microprocessor provides four execution or access levels for processes. This allows each process to have functions that operate at different levels to provide the proper levels of system protection. These levels range from the *most privileged* (level 0) to the *least privileged* (level 3). Through built-in microprocessor safeguards, the privilege level serves as a protection level. One of the functions of the MMU is to ensure that code and data in any particular level are accessed only by code or processes that have the right permissions. The four execution levels are defined as:

- Kernel (level 0) - The most privileged level; it contains the operating system's most privileged services (e.g., device drivers and interrupt handlers).
- Executive (level 1) - This level is provided for greater flexibility in the operating system design.
- Supervisor (level 2) - Common library routines can operate at this level and be safe from corruption by the level 3 activities.
- User (level 3) - The least privileged level; most user programs can run in this level.

Table 4-1 lists the powerful *WE* 32100 Microprocessor instructions provided for operating systems. These instructions have two levels of hierarchy: *privileged* and *nonprivileged.* Privileged instructions may be executed only if the processor is in kernel level, and they are used to perform process switches, to enable or disable the MMU, or to suspend fetching of instructions. Nonprivileged instructions do not depend on the execution level (i.e., they can be executed at any level) and are used to switch between execution levels (in ways restricted by the operating system) or to convert a virtual address to a physical address.

The processor automatically executes the appropriate *microsequence* (a built-in sequence of actions) when an interrupt is requested or an exception occurs. These microsequences and many operating system instructions can call functions (also microsequences) that do the context switching (changing the hardware context for the new process to be executed). This feature takes the requirements of context switching out of the operating system, allowing for quicker and more efficient operating system design and execution. The operating system instructions and microsequences are described in **4.8 OPERATING SYSTEM INSTRUCTIONS.**

Table 4-1. Operating System Instructions

Privileged Instructions			
Instruction	**Assembly Syntax**	**Hex Opcode**	**Description**
Enable virtual pin and jump	ENBVJMP	300D	Enables the MMU to translate addresses. The virtual address of the first instruction to be executed after the MMU is enabled must be stored in register **r0** before this instruction is executed.
Disable virtual pin and jump	DISVJMP	3013	Disables the MMU from translating addresses. The physical address of the first instruction to be executed after the MMU is disabled must be stored in register **r0** before this instruction is executed.
Call Process	CALLPS	30AC	Performs an explicit process switch.
Return to process	RETPS	30C8	Restores a process from an interrupted state.
Wait for interrupt	WAIT	2F	Stops the CPU from fetching instructions. Fetching resumes after an interrupt is encountered.
Interrupt Acknowledge	INTACK	302F	Stores interrupt id in **r0**.
Move translated word	MOVTRW *src,dst*	0C	The MMU converts the virtual address specified by *src* to a physical address. The result is stored in *dst*. Can be used to obtain physical address to send to an I/O device.

Table 4-1. Operating System Instructions (Continued)			
Nonprivileged Instructions			
Instruction	**Assembly Syntax**	**Hex Opcode**	**Description**
Gate	GATE	3061	Mechanism used to transfer control between different execution levels.
Return from Gate	RETG	3045	Returns control to the function which called the gate. Linear ordering of execution levels is enforced by RETG (i.e., new execution level may not be more privileged than the current level).

Other features of the microprocessor's architecture that are provided for operating system design are summarized as follows:

- The microprocessor supports different levels of execution privilege and enforces linear ordering of these levels only on a return-from-gate (RETG) instruction, as discussed in **4.3.3 Return-From-Gate Instruction**.
- The microprocessor provides flexibility in transferring execution control between privilege levels. Control is transferred through the gate mechanism, as discussed in **4.3 SYSTEM CALL**.
- A scheduler may explicitly switch processes (CALLPS or RETPS instructions), but part of the interrupt structure and certain exception conditions involve implicit switching of processes. This provides some of the interrupt structure and some of the exception handler advantages of a process switch.
- The processor supports a layered exception-handling structure that uses different mechanisms (process switching or gate mechanism), depending on the severity of the exception.
- The processor supports *full* and *quick* interrupt handlers that use different mechanisms (process switching or gate mechanism). A full interrupt is handled as an implicit process switch, while a quick interrupt is handled as an implicit gate. See **2.8 INTERRUPTS** for details on determining how interrupts are to be handled (i.e., as full or quick interrupts).
- Address space of each process may include the space that contains the operating system; i.e., the user may pass and address arguments across system calls efficiently, but need not switch memory map information across such calls.
- The processor supports memory management, permitting users to believe the system has 4 Gbytes of memory. However, the operating system must provide the information required by a memory management unit (MMU) to translate virtual addresses (i.e., memory descriptors) or disable the MMU for physical addressing. Systems without an MMU use only physical addressing.

4.1.1 Memory Management Considerations for Virtual Memory Systems

A memory management unit (MMU) is required for virtual memory (storage) systems. The primary function of an MMU is to translate virtual address into physical addresses and implement the protection of each process' data. The features that support a virtual memory operating system are:

- Support of contiguous segments and paged segments. Segments, or blocks of memory, are defined by memory descriptors. The *WE* 32101 Memory Management Unit uses segment descriptors to define contiguous segments (i.e., a block of memory defined up to 128 Kbytes in length) and segment and page descriptors to define paged segments (i.e., a block of memory defined to contain up to sixty-four 2 Kbyte pages).
- *Present* bits to indicate whether or not a segment is currently in main memory.
- *Referenced* and *modified* bits to aid implementation of a least recently used (LRU) algorithm in the operating system.
- An *indirection* feature that allows segments to be given different access permissions (e.g., read or write), yet still be shared by different routines running at the same execution level (see **4.7.5 Indirect Segment Descriptors**).
- Access fields contained in segment descriptors are used to provide protection so that segments are accessed in the appropriate way by the appropriate execution level. An access exception is generated if access is disallowed.
- An *object-trap* feature provides a mechanism where I/O devices or external processors appear as normal segments from the user-software point of view.
- Segment marking as cacheable or not cacheable using a cacheable bit. This can be used to aid the use of an external data cache in the system main memory (see **4.7.6 Using the Cacheable Bit**).
- A unique exception (page-write) that can be issued on any attempt to write a given page (see **4.7.7 Using the Page-Write Fault**).

4.2 STRUCTURE OF A PROCESS

Each process executing in the *WE* 32100 Microprocessor consists of the following elements:

- A processor status word (PSW) - the CPU register that contains status information about the instruction just executed and the current process.
- A process control block (PCB) - a process data structure in external memory that contains the hardware context of a process when the process is not running. This context consists of the initial and current contents of control registers; PSW, program counter (PC), and stack pointer (SP); the last contents of the general-purpose registers r0 through r8, frame pointer (FP), and argument pointer (AP); boundaries for an execution stack; and block-move specifications for the process.

- A process control block pointer (PCBP) - the CPU register that identifies the starting location of the PCB for the process currently executing.
- Memory address space (the areas in memory allocated for the process). This space can be defined by memory management specifications in the PCB block-move area.
- Segment and page descriptors and MMU SRAMs register contents, if the system uses an MMU. This information can be defined in the PCB block-move area for automatic transfer to the MMU during a process switch.

4.2.1 Execution Privilege

As stated previously, the processor recognizes four execution modes: kernel (most privileged level), executive, supervisor, and user (least privileged level). Controlled entry to an execution mode does not assume a particular order of the levels, but controlled return does. Controlled return enforces a four-level privilege hierarchy going from most privileged to least privileged; from kernel to executive to supervisor to user. See **4.3 SYSTEM CALL** for a description of controlled transfers across privilege levels. The operating system design may use the four execution modes to manage layers of control. However, further protection for memory access must be built into a memory management system.

To protect against an unwanted process switch, privileged operating system instructions may be executed only in kernel mode. The other operating system instructions and the instruction set may be executed in any of the four modes. Thus, only a two-level privilege hierarchy exists for instruction execution.

Information associated with a process is protected by the restriction that the processor be in kernel mode when writing the following registers:

1. Processor status word (PSW) - provides information about the current process. The microprocessor implicitly alters the condition flags after most instructions. In addition, some PSW fields change their contents to identify the type and severity of an exception and help the operating system select the appropriate exception handler.
2. Process control block pointer (PCBP) - contains the starting address of the PCB for the current process. Because the PCB for a process is assigned to a fixed starting location, the PCBP content changes only during a process switch.
3. Interrupt stack pointer (ISP) - points to a stack which is used to store the PCBP for interrupted processes and restores the PCBP when a process returns from its interrupted state. Generally, the ISP is altered only on a process switch.

If the processor is not in kernel mode, it generates a normal exception (privileged register) when an instruction tries to write to one of them. The use of privileged registers is discussed later.

4.2.2 Execution Stack

During the execution of a process, the CPU SP register identifies the address of the next available location on an execution stack. Conventionally, such a stack could be used for

linking functions and passing arguments between:

- Functions that execute at the same level
- A privileged function and its less privileged caller
- An exception handler and the function that caused the exception
- An interrupt handler and the interrupted function.

An execution stack also provides temporary storage for local variables.

Unlike other architectures that require at least two stacks, the *WE* 32100 Microprocessor has only one execution stack per process. In some other processors, one stack serves the most privileged execution levels, while the other is used in less privileged levels. Other processors generally use a stack for each privilege level. A privileged stack in other architectures is protected from errors in less privileged levels that could destroy its contents.

In the *WE* 32100 Microprocessor architecture, a process uses one stack in all execution modes. Each process stack is protected through maintenance of its upper and lower bounds in the process control block (the data area that stores the hardware context) for the process and checking of the bounds during a gate operation. Thus, each execution level is protected from stack errors by other execution levels. In addition, using only one stack reduces the overhead for stack allocation and simplifies the management of process stacks.

Before executing a transfer to a more privileged level through a *system call or gate*, the processor checks the current SP against the stack bounds. The transfer occurs if the SP falls within bounds. Otherwise, a stack exception (stack-bound) is generated.

Using the execution stack for the process, the processor handles normal exceptions within the process in which they occurred. Before transferring to the appropriate exception handler, it checks the current SP against the stack bounds.

Because an interrupt other than a quick interrupt causes a process switch, the processor interrupt structure uses a different execution stack for each interrupt handler. Therefore, the sanity of the interrupted process execution stack does not have to be checked. In addition, the processor stores the PCBP of each interrupted process on one system-wide interrupt stack and retrieves it from that stack when the process resumes execution. Quick interrupts save the PC and PSW context on the execution stack of the active process and are handled in the same manner as normal exception.

4.2.3 Process Control Block

Each process has a process control block (PCB). Elements in the PCB are accessed through the process control block pointer (PCBP). This privileged register contains the starting address in memory of the PCB for the process that is currently executing. Although PCBs can be stored anywhere in memory, Table 4-2 identifies where the PCBP must be stored for various processes.

Table 4-2. PCBP Locations	
Process	**Location (See note)**
Full interrupt handler	Vectors start at location 140 (0x8C). Each interrupting device has an 8-bit Interrupt-ID; the PCBP for the appropriate interrupt handler should be at location 140+4* Interrupt_ID.
Reset exception	Physical location 128 (0x80)
Process exception	Location 132 (0x84)
Stack exception	Location 136 (0x88)
Call process (CALLPS)	PCBP taken from register r0; must be stored in r0 before CALLPS instruction is executed
Return to process (RETPS)	PCBP taken from top of interrupt stack when RETPS instruction is executed

Note: All locations are virtual addresses in virtual mode and physical addresses in physical mode. Locations are given as decimal values (hexadecimal values).

Because only one process executes at a time in a multiprogramming system, the PCB of a process retains its hardware context when that process is not running. The PCB, illustrated on Figure 4-1, contains:

- Initial context. The three control registers (PC, PSW, and SP) are loaded with initial values when a process starts executing for the first time. First time execution is indicated by the I bit in the PSW being set (1).
- Control register save area. When a process is interrupted, the current contents of its control registers are saved here. These values are loaded when that process resumes execution and the I bit in the PSW is clear (0).

Note: If the I bit in the PSW of a process is initially set (1), execution starts from its initial-context values. If the bit is clear (0), execution resumes from an intermediate context. See **4.4.1 Context Switching Strategy** for more information on the I bit.

- Stack bounds. The upper and lower stack bounds define the area allocated to the execution stack for this process.
- General register save area. This area is reserved for saving the contents of register r0 through r10. Registers r9 and r10 are the frame pointer (FP) and argument pointer (AP), respectively. These are used to specify the location of variables or arguments. The FP locates local variables for a function, while the AP locates arguments passed to the function.
- One or more block-move areas. If a process does not require any block moves (usually used to perform a change in memory management specifications), only the null block is required in the PCB. Otherwise, it contains a block-move area for each move to be performed.

Note: The R bit of the PSW must be set (1) if the general registers are to be saved for the old process or loaded for the new process and if the block moves are to be executed for the new process. See **4.4.1 Context Switching Strategy** for more information on the R bit.

In general, the PC and SP values and block addresses stored in a PCB may be physical or virtual addresses. If they are virtual addresses, the MMU must be enabled to translate them into physical addresses. Two operating system instructions, enable virtual pin and jump (ENBVJMP) and disable virtual pin and jump (DISVJMP), enable or disable the processor's virtual address pin to tell the MMU it is generating virtual (enable) or physical (disable) addresses. Before the instruction ENBVJMP or DISVJMP is executed, the virtual or physical, respectively, address of next instruction to be executed must be stored in r0.

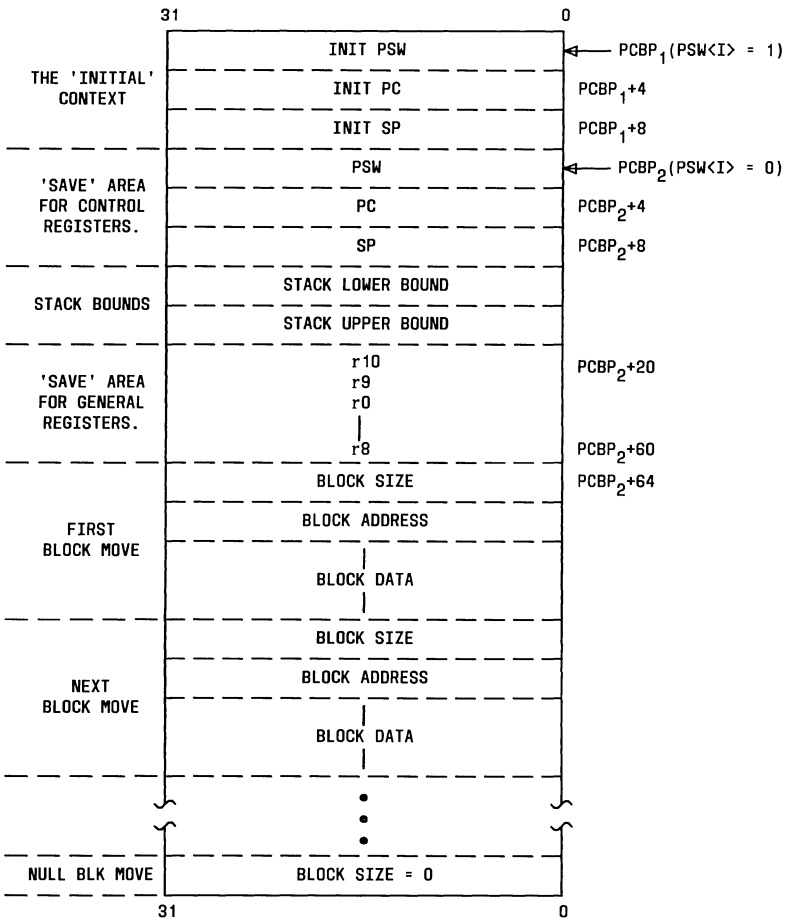


Figure 4-1. A Typical Process Control Block

Initial Context for a Process

The initial context of the executing process is set up as follows:

- The PCBP, stored in memory, points to the initial-context area of its PCB.
- The initial PSW occupies the first PCB location, and its I bit should be set (1) to identify that the process starts executing from its initial context. (The R bit should be set if this process will use general registers.) See **4.2.4 Processor Status Word** and **4.4.1 Context Switching Strategy** for more details about the R and I bits.
- The second PCB word, the initial PC, is the address of the first instruction that process executes.
- The third PCB word contains the initial SP (the address of the first location on the execution stack).
- The seventh and eighth PCB words define the upper and lower limits of the execution stack.

The values in the initial-context area and the stack bounds never change during normal execution.

Saved Context for a Process

When a process switch occurs, the processor uses the current PCBP to save the context of Process A (the executing process) in the current PCB. Using offsets from PCBP to access the correct PCB location for Process A, the processor stores PC, PSW, SP, and if the PSW R bit is set to 1, the general registers. It then reads in a new PCBP value for Process B (the incoming process) and loads the Process B context from its PCB.

Memory Specifications

On each process switch, if the R bit in the PSW is set (1), the processor, using information in the process PCB, performs a series of block moves. The PCB provides three elements for each block move (see Figure 4-1):

1. Block size - This word value specifies the length of the block (number of words to be moved) and implicitly identifies the starting location of the next block-move area.
2. Block address - This word value is the destination address where the processor starts writing the block data.
3. Block data - This series of words represents the data to be moved. If the system has an MMU, it could be the information written to MMU registers (or tables) to set up the memory context for the new process.

The processor executes a move block (MOVBLW) instruction for each block until a zero-length block (Block size = 0) is reached.

A memory management scheme does not alter the way the processor performs the block moves or how many block moves occur. However, memory management may affect block addresses. Systems with an MMU should use a virtual address for each block when the MMU is enabled and physical addresses when the MMU is disabled. For a system without an MMU, a block address must be a physical address.

4.2.4 Processor Status Word

The processor maintains a 32-bit processor status word (PSW) register which defines the state of a currently running process. Table 4-3 identifies its contents.

The read-only fields of the PSW cannot be altered by software regardless of the execution mode. An exception or process switch always directly affects the ET, ISC, and TM fields. The ET and ISC fields, which identify the type and cause of an exception, are part of the exception mechanism described in **4.6 EXCEPTIONS**. The TE and TM fields are part of the trace-trap mechanism.

An instruction may read the PSW at any time, but may write it explicitly only when the process is in kernel mode. However, the processor implicitly alters some fields during normal execution at other levels. In particular, most instructions change the condition flags.

4.3 SYSTEM CALL

The system-call (*gate*) mechanism provides a means of controlled entry into a function by installing a new PSW and PC value. If the new PSW has a different privilege level than the current PSW, a transition to a different execution level occurs.

On simpler processors, a trap or supervisor call instruction picks up a new PC and PSW from a fixed location. Then the software has to perform further indirection based on the "trap number." The gate mechanism, embodied in its gate (GATE) instruction, automatically performs this second level of indirection for the user. The gate mechanism is described **4.3.1 Gate Mechanisms**.

Table 4-3. Processor Status Word Fields

Bit(s)	Field	Contents	Description
0—1	ET	Exception Type	This read-only field indicates the type of exception generated during operations and is interpreted as: **Code Description** 00 On Reset Exception 01 On Process Exception 10 On Stack Exception 11 On Normal Exception
2	TM	Trace Mask	The read-only TM field enables masking of a trace trap. This bit masks the trace enable (TE) bit for the duration of one instruction to avoid a trace trap. The TM bit is set (1) at the beginning of every instruction and cleared (0) as part of every microsequence that performs a context switch or a return from gate.
3—6	ISC	Internal State Code	This 4-bit code distinguishes between exceptions of the same exception type. The ISC is a read-only field.
7—8	RI	Register-Initial Context	These bits control the context switching strategy. The I bit (bit 7) determines if a process executes from initial or intermediate context. The R bit (bit 8, read only) determines if the registers of a process should be saved. It also controls block moves to change map information.
9—10	PM	Previous Execution Level	This field defines the previous execution level. The code is interpreted as: **Code Description** 00 Kernel level 01 Executive level 10 Supervisor level 11 User level
11—12	CM	Current Execution Level	This field defines the current execution level. The CM code is interpreted the same way as the PM code. Changes to the CM field via instructions with the PSW as an explicit destination may cause the XMD pins to change in the middle of a memory access, which could cause a spurious exception or system problem. Therefore, only microsequence instructions should be used to change the CM field state.

Table 4-3. Processor Status Word Fields (Continued)

Bit(s)	Field	Contents	Description
13—16	IPL	Interrupt Priority Level	The IPL field represents the current interrupt priority level. Fifteen levels of interrupts are available. An interrupt, unless it is a nonmaskable interrupt, must have a higher priority level than the current IPL in order to be acknowledged. Therefore, level 0000 indicates that any of the fifteen interrupt priority levels (0001 through 1111) can interrupt the microprocessor; level 1111, the highest interrupt priority level, indicates that no interrupts (except a nonmaskable interrupt) can interrupt the microprocessor.
17	TE	Trace Enable	This bit enables the trace function. When TE is set (1), it causes a trace trap to occur after execution of the next instruction. Debugging and analysis software use this facility for single-stepping a program. Changes to the state of the TE bit via instructions with the PSW as an explicit destination may cause unpredictable trace behavior. Therefore, only microsequence instructions should be used to change the TE bit state.
18—21	NZVC	Condition Codes	The condition codes reflect the resulting status of the most recent instruction execution that affects them. These codes are tested using the conditional branch instructions and indicate the following when set (1): N - Negative (bit 21) V - Overflow (bit 19) Z - Zero (bit 20) C - Carry (bit 18)
22	OE	Enable Overflow Trap	This bit enables overflow traps. It is cleared (0) whenever an overflow trap is detected and handled.
23	CD	Cache Disable	This bit enables and disables the instruction cache. When the CD bit is set (1), the cache is not used. Changes to the state of the CD bit via instructions with the PSW as an explicit destination may corrupt the contents of the instruction cache. Therefore, only microsequence instructions should be used to change the CD bit state.
24	QIE	Quick-Interrupt Enable	The QIE enables and disables the quick-interrupt facility. If QIE is set (1), an interrupt is handled via the quick-interrupt sequence.
25	CFD	Cache Flush Disable	When this bit is set (1), it disables cache flushing (emptying of the instruction cache contents) during the XSWITCH_TWO microsequence.
26—31		Unused	These bits are not used and are always cleared (0).

4.3.1 Gate Mechanism

The CPU contains a microsequence program that locates the handling routine for the gate mechanism. To use this mechanism, the operating system must provide the following gate mechanism tables:

- Pointer table - Contains the 32-bit starting addresses for a set of handling-routine tables. The processor assumes address 0 as the beginning of the table. The table contains thirty-two 4-byte (word) addresses, one for each handling-routine table.

Note: Use of kernel level is forced whenever this table is accessed during execution of the GATE instruction.

- Handling-routine tables - Each table in the set contains the entry points (PSW and PC values) for a group of functions. A table is limited to 4096 two-word entries; one a new PSW and the other a new PC (in that order) for a controlled transfer.

Two indexes, obtained from a GATE instruction's implied operands, locate the appropriate PC and PSW pair for the controlled transfer.

Pointer Table

This table contains thirty-two entries and starts at location 0. It must be contained in secure memory (write permission for kernel level only) to prevent unwarranted access. The first entry is reserved for normal-exception handling. Therefore, address 0 must locate the handling-routine table (entry point set) for the normal-exception handlers.

The rest of the addresses in the pointer table may define sets of entry points for controlled transfers. For example, one entry can be used to locate the handling-routine table for kernel level entries, one entry for executive level entries, one for supervisor level entries, and one for user level entries.

All thirty-two entries in the pointer table must be defined. A typical use for the remaining entries is to define all unused pointer table entries to point to a dummy handling-routine table. The dummy table is typically used to prevent an exception from occurring should an offset into the pointer table result in locating an undefined handling-routine table.

Handling-Routine Tables

A handling-routine table stores a maximum of 4096 entry points (PSW and PC pairs) and may be placed anywhere in memory (virtual memory if the system has an MMU that is enabled; physical memory if it does not). However, each must start at an address that is a multiple of eight. In a typical system, the handling-routine tables for entry into kernel level reside in a section of memory that is shared by all processes.

Note: Sections of memory do not imply execution level. The GATE instruction forces kernel level before it accesses any handling-routine tables. To preserve table security, these tables should be protected so only the kernel level can write to them.

4.3.2 GATE Instruction

The GATE instruction is modeled after the jump to subroutine (JSB) instruction rather than the call procedure (CALL) instruction which calls a function. In the typical system environment (e.g., *UNIX* System, C compiler), the compiler generates a call to an assembly-language function which then executes the gate instruction. GATE needs only to execute a simple jump since the 'call frame' already exists.

Although GATE may be executed at any privilege level, the CPU forces and releases kernel level for memory access. The gate instruction has two entry points. GATE starts execution at the first entry point, while the on-normal exception microsequence enters at the second (see **4.6 EXCEPTIONS**). The second entry point is also the start of the gate mechanism.

Before a GATE instruction is executed, two registers must be loaded:

- Register 0 (r0) must be loaded with the offset for constructing *index1* (the index into the pointer table). Index1 identifies the starting address of the appropriate handling-routine table. Only five bits of r0 are used.
- Register 1 (r1) must be loaded with the offset for constructing *index2* (the index into the handling-routine table). Index2 locates the new PSW and PC.

The on-normal exception microsequence is modelled after a GATE. On a normal exception, the CPU supplies all appropriate information needed to execute a GATE-like sequence.

The GATE instruction executes the following tasks in sequence (see Figure 4-2).

First Entry Point

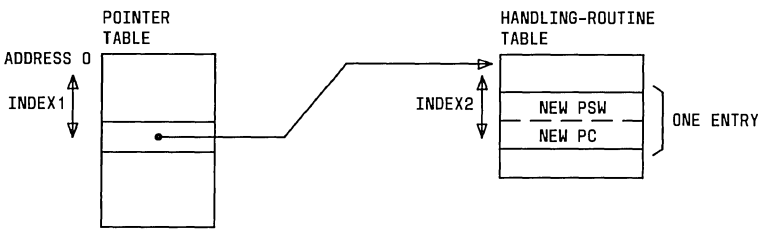
1. GATE forces kernel level on memory accesses and checks the current SP against the upper- and lower-stack bounds in the currently executing process PCB. A memory exception on accessing either of the stack bounds from the PCB causes a process exception (GATE-PCB). If SP is outside either boundary, a stack exception (stack bound) is generated. GATE then releases kernel level for memory accesses.
2. GATE writes 1, 0, 2 to the ISC, TM, and ET fields, respectively, of PSW. Then it saves the address of the next instruction (PC + 2) and the current PSW on the execution stack. If a memory exception occurs on the stack accesses, the processor generates a stack exception (stack).
3. GATE computes index1 for the pointer table by masking the contents of r0 with 0x7C and places the result in tempa. It then masks the contents of r1 with 0x7FF8 for index2 and stores the result in tempb. (Special registers tempa and tempb are used in later steps for accessing the handling-routine tables.)

Second Entry Point - The Gate Mechanism

1. GATE again forces kernel execution level for memory accesses.
2. GATE uses tempa as a pointer to read the starting address of a handling-routine table from the pointer table and write it to tempa. It then adds tempa and tempb (the offset into the handling-routine table) and stores the result, index2, in tempb. This is the address of the new PSW and entry point PC for the GATE jump.
3. GATE uses index2 to get new values for PSW fields OE, NZVC, TE, CM, R, and I. It then sets PSW fields ISC, TM, and ET to 7, 0, and 3, respectively.
4. GATE uses index2 to locate and load the new PC.
5. GATE adjusts SP to a location above the saved PC and PSW (thus completing a push of the PC and PSW onto the stack) and releases kernel level for memory accesses.

The processor then begins executing the handling routine. When the routine finishes, a return from gate (RETG) instruction returns to the function that issued the system call.

Note: If the GATE instruction is invoked directly, a memory exception that occurs during the remaining steps causes a normal exception (gate-vector). A normal-exception microsequence entering here will already have kernel level in effect and values in tempa and tempb. Entering at this point from a normal-exception microsequence means that a memory exception for any step generates a reset exception (gate-vector).



index1 = r0 & 0x7C
index2 = r1 & 0x7FF8
Entry Address for Handler Routine = (Address Pointed to by index1) + index2

Figure 4-2. Tables for the Gate Mechanism

4.3.3 Return-From-Gate Instruction

The return-from-gate (RETG) instruction is modeled after a return-from-subroutine (RSB) instruction rather than after a return-from-procedure (RET) instruction. Unlike the gate instruction, RETG enforces linear ordering of execution levels, which means the new execution level may not be more privileged than the current level. During an RETG, the microsequence forces and releases kernel level as required for memory access.

The return-from-gate instruction performs the following sequential actions to return to the calling function.

1. Retrieves the old PSW and next-instruction address (stored on the execution stack by the corresponding GATE) and places these in tempa and tempb, respectively.
2. Sets the trace mask (TM) bit in PSW to zero.
3. Compares the CM field in the current PSW to the CM field of the old PSW (in tempa) to verify that the new execution level is less than or equal to the current level. If this test fails, the microprocessor issues a normal exception (illegal-level change).
4. Writes the PSW fields OE, NZVC, TE, CM, PM, R, and I using the values in tempa (the saved PSW).
5. Loads PC from tempb.
6. Adjusts SP to the location below the saved PSW and PC (thus completing a pop of the PSW and PC from the stack).
7. Writes 7, 0, and 3 to PSW fields ISC, TM, and ET, respectively.

The function that called the GATE then starts executing its next instruction.

Note: If a memory exception occurs on a stack access during these steps, a stack exception is issued.

4.4 PROCESS SWITCHING

Using its PCB, the *WE* 32100 Microprocessor explicitly invokes a process by automatically saving or restoring its context. However, a PCB only defines hardware context (as described in **4.2.3 Process Control Block**), not software-maintained information (i.e., variables and arguments pointed to by the argument pointer and frame pointer) for the process. The PCBP register always contains the address of the PCB of the current process.

To avoid destroying the PCB content on a process switch, the call process (CALLPS) instruction performs both the save of the previous context and load of the new process context. The processor does not accept interrupts until the CALLPS instruction is completed. This prevents an undefined state between a save and a load. In this state, a PCBP would still point to the PCB for the old (exiting) process. If the system completes a save just as an interrupt occurs, then the interrupt-handling scheme causes the saved PCB context to be overwritten. This cannot happen with the *WE* 32100 Microprocessor.

4.4.1 Context Switching Strategy

The process-switch mechanism uses two PSW parameter bits, R and I, to control the context-switching strategy:

- The R bit determines if the CPU general registers used by a process should be saved. It also controls block moves.
- The I bit determines if a process executes from an initial context or intermediate context. It also affects the setting of the PCBP register.

To save or load the appropriate information on a process switch, the processor uses the R and I bits in the PSW of the new or incoming process. The use of the R and I bits is explained next.

R Bit

The use of the R bit is explained by considering two processes: Process A as the current or old process, and Process B as an incoming process. If Process B's PSW R bit is set, this signifies that Process B wants to use the general registers, and thus the CPU's general registers are saved in Process A's PCB save area for general registers when the process switch occurs. Later, on return to Process A, the general registers will be restored for Process A. If Process B requires block moves, the R bit must be set. On a process switch, where a CALLPS (call process instruction) or simulated CALLPS is performed, the processor saves general registers for Process A and performs block moves contained in Process B if the R bit of Process B's PSW is set. When a process switch occurs as a result of the RETPS instruction, the general registers are restored if Process A's PSW R bit is set. (This value was copied from Process B's PSW when CALLPS occurred.)

To generalize, set the R bit in the initial-context PSW of any process that uses the general registers or requires block moves. The R bit setting never changes, even though a process switches in and out many times.

I Bit

The I bit function identifies whether a process is to start from an initial or intermediate context. It also affects the PCBP register.

Consider two processes: Process A, the current or old process, and Process B, the incoming or new process. The function of the I bit is explained as follows:

- On leaving Process A, the microprocessor always writes the PC, SP, and PSW values starting at the location pointed to by Process A's PCBP and then saves Process A's PCBP on the interrupt stack. On entry to Process B, the microprocessor always reads the PSW, PC, and SP values starting from the location pointed to by the Process B's PCBP. These operations are the same for the CALLPS instruction, full interrupts, and exceptions that perform a process switch.

- If the I bit is set (1) in Process B's PSW, Process B's PCBP is incremented by twelve bytes (three words) after the PSW, PC, and SP are loaded, and the I bit is set to zero. Incrementing the PCBP guarantees that the initial context loaded in the first step will not be overwritten if Process B is interrupted or executes a CALLPS instruction. Clearing the I bit ensures that the adjustment of the PCBP is done only once. (If this was not done and the I bit was to remain set, and if Process B was repeatedly interrupted and resumed, Process B's PCBP would be incremented by twelve on each RETPS instruction.)
- When Process B executes a RETPS instruction, Process A's PC, SP, and PSW context is loaded from the locations pointed to by PCBP popped off the interrupt stack.

The main idea is that the effect of the I bit of a given process is not seen until that process is itself interrupted and then returned to by another process.

If the I bit of a process is set when it is entered initially, the process' initial context will be preserved if it is interrupted or if it calls another process. The saved context will be written to and retrieved from the twelve bytes adjacent to the initial context. Otherwise, if the I bit is zero initially, the initial context (if writable) will be overwritten in the course of servicing the interrupt or CALLPS instruction.

Another way to look at the I bit is that if the PSW I bit feature did not exist, and the user wanted to modify the PCBP via software to save the initial process context, it could not be guaranteed that the PCBP would be adjusted before another interrupt was taken. Since the I bit adjustment is done in a CPU microsequence, it guarantees that the PCBP adjustment is made while the CPU is immune to interrupts.

The following describes the effects on the PCBP and the initial- and saved-context areas during process switches.

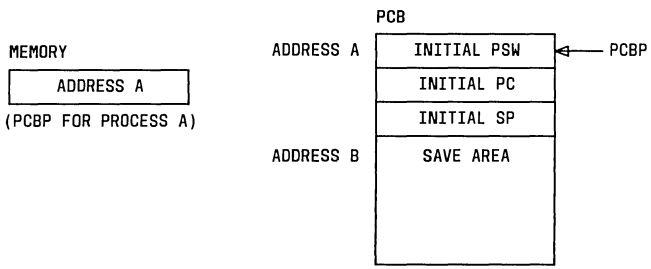
When Process A is called initially by the CALLPS instruction (an explicit process switch), the processor loads the PCBP register with the starting address (Address A) of the Process A PCB (see part A of Figure 4-3). It then loads the PSW, the program counter (PC), and the stack pointer (SP) with their initial context. Next, if the I bit in the PSW is set (1), the processor clears the I bit and increments the PCBP register by twelve bytes to the saved-context area (Address B) of the Process A PCB (see part B of Figure 4-3). This will cause any later process switch to save PSW, PC, and SP values in the intermediate context area instead of overwriting the initial-context values. The Process A initial-context area and its PCBP stored in memory are not affected on this process switch.

Part A of Figure 4-4 shows the effect on the PCBP and the Process A PCB if a process switch occurs before Process A is finished. Here, the processor uses the adjusted PCBP (assuming the I bit was set when Process A was initiated) to save the intermediate context of the control registers and stores the PCBP on the interrupt stack. This time, the PSW I bit will be clear and the PC points to the next Process A instruction.

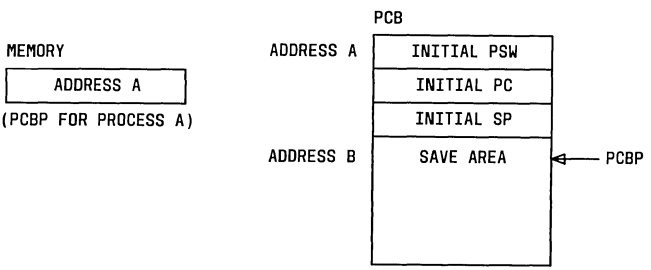
When the processor restores Process A (see part B of Figure 4-4), the processor retrieves the PCBP from the interrupt stack. Remember that the PCBP points to the saved-context

area (if the initial I bit value was zero, then the saved-context area overwrote the initial-context area) and the I bit of the PSW is clear. The processor then loads the control registers with their intermediate context and Process A resumes execution with its next instruction. If the initial value of the I bit for Process A was clear (0), then the initial-context area becomes the save area since the PCBP was never adjusted to point to the saved-context area. That is, the Address B in Figures 4-3 and 4-4 is the same as Address A, and the initial-context area no longer exists.

The initial context of a process never changes, provided the initial I bit setting is one. Also, the PCBP stored in memory always points to the initial context. This enables an interrupt-handler process to get its PCBP from memory without going through a scheduler. A suspended process restarts from an intermediate context on a return from a full-interrupt handler, certain exception handlers, or the RETPS (return-to-process) instruction. Also, a process that had an initial I bit value of zero is restarted from an intermediate context on any subsequent CALLPS instruction after it was first switched to. A process starts from its initial context (initial I bit value is set) whenever a CALLPS instruction is executed.

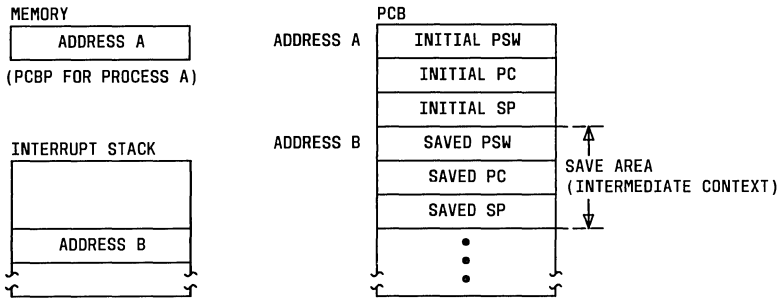


A. Context at Start of Switch to Process A

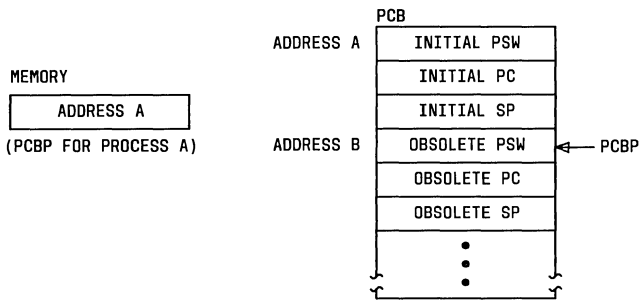


B. Context After Switch to Process A

Figure 4-3. A PCB on an Initial Process Switch to a Process



A. Context After Switch to Some Other Process



B. Context After Process A Is Switched Back to and Restored

Figure 4-4. A PCB on a Process Switch During Execution of a Process

4.4.2 Call Process Instruction

The call process (CALLPS) instruction, mentioned in the discussion of the R and I bits, is the process analog of the call procedure (CALL) and save registers (SAVE) instructions that carry out a function call. To execute CALLPS, the processor must be in kernel mode. In addition, r0 must be preloaded with the new PCBP (address of the PCB for the new process).

The call process instruction performs an explicit process switch. Using Process A as the current (old) process and Process B as the incoming (new) process, CALLPS performs the following sequential steps:

1. Places the content of r0 (Process B PCBP) into register tempa and forces kernel execution level on memory accesses.
2. Saves Process A PCBP on the interrupt stack (see **Interrupt Stack and ISP** under **4.5.2 Interrupt Mechanism**). If a memory exception occurs when accessing this stack, the processor issues a reset exception (interrupt stack).
3. Adjusts PC to the address of the instruction that Process A would have executed next (PC + 2).
4. Calls the function XSWITCH_ONE() to save Process A context. (All writes are made to the saved-context area of process PCB because the I bit of an executing process PSW is always clear.) If a memory exception occurs on a PCB access, the processor issues a process exception (old PCB).

   XSWITCH_ONE does the following:

   a. Using tempa as a pointer to the Process B PCB, copies the R bit from the new PSW into the R bit of the current PSW. (The R bit will be used later.)
   b. Stores the current PSW in the Process A PCB and writes 0, 0, 1 to the ISC, TM, and ET fields, respectively, of the saved PSW.
   c. Saves PC (address of the next instruction) and SP in the Process A PCB.
   d. Writes r0 through r10 to the general register area of the Process A PCB if the R bit of the Process B PSW is set. Otherwise, these registers are not saved.
   e. Returns control to CALLPS.

5. Calls the function XSWITCH_TWO() to load the Process B context. If a memory exception occurs when accessing its PCB, the processor issues a process exception (new PCB).

   XSWITCH_TWO does the following:

   a. Loads PCBP from tempa (which contains Process B's PCBP value).
   b. Reads in the new PSW and sets its TM bit to 0. Next, it loads the new PC and SP. PC now contains the address of the first instruction for Process B.
   c. Tests the PSW I bit. If the I bit is set, the I bit is cleared, and the PCBP is adjusted to the saved-context area of the Process B PCB.
   d. Returns control to CALLPS.

6. Writes 7, 0, 3 to the ISC, TM, and ET fields, respectively, of the PSW.
7. Calls the function XSWITCH_THREE() for block moves.

   XSWITCH_THREE does the following:

   a. Tests the R bit in the PSW.

      - If the R bit is set, it loads the block-move information from the block-move areas of the Process B PCB. For each block to be moved, it preloads r0 with the starting address of the block-move area, r1 with the size of the block (number of words to be moved), and r2 with the destination of the move. Then it executes a move block (MOVBLW) instruction.
      - If the R bit is clear (0), no block moves are performed.

   b. Returns control to CALLPS.
8. Releases kernel execution level on memory accesses and Process B begins executing.

4.4.3 Return-to-Process Instruction

The RETPS instruction restores a process from its interrupted state and may be executed only when the processor is in kernel mode. RETPS is the process analog of a function return that uses the restore registers (RESTORE) and return-from-procedure (RET) instructions. Again, the R and I bits in the PSW determine the context-switching strategy.

The CALLPS and RETPS instructions act similarly, except the RETPS does not save the context of the exiting process. For this discussion, Process A is the returned-to-process. RETPS performs the following sequential steps:

1. Forces kernel execution level on memory access and moves the Process A PCBP from the interrupt stack into register tempa. If a memory exception occurs on the stack access, the processor issues a reset exception (interrupt-stack).
2. Loads the PSW R bit with R bit from tempa.
3. Calls XSWITCH_TWO() to restore the Process A context. If a memory exception occurs when accessing its PCB, process exception (new PCB) is issued. (The PCBP for Process A is still at the top of the interrupt stack.)

   XSWITCH_TWO does the following:

   a. Loads PCBP from tempa.
   b. Loads PSW from the PCB, writes a 0 to the TM bit, and then loads PC and SP. Because this is a return process, the I bit is clear and all control registers are loaded from the saved-context area of its PCB.
   c. Returns control to RETPS.
4. Writes 7, 0, 3 to the ISC, TM, and ET fields, respectively, of PSW.
5. If R bit is set (1), calls XSWITCH_THREE() to perform any block moves.

XSWITCH_THREE does the following:

a. Tests the R bit in the PSW.

- If the R bit is set (1), it does the block moves in the block-move areas of the Process A PCB. For each block to be moved, r0 gets the starting address of a block-move area in the PCB, r1 gets the size of the block (number of words to be moved), and r2 gets the destination of the move. Then the function executes a move block instruction (MOVBLW).
- If the R bit is clear (0), no block moves are performed.

b. Returns control to RETPS.

6. If the R bit is set (1), RETPS loads r0—r10 from general register save area of Process A PCB.
7. Releases kernel execution level on memory accesses and Process A resumes executing.

4.5 INTERRUPTS

When an external device requests an interrupt, a processor temporarily stops its current execution and jumps to code that services the interrupt. On completion of the interrupt handler code, execution resumes at the point where the interrupt occurred. An interrupt mechanism performs the execution switch.

4.5.1 Interrupt-Handler Model

An interrupt handler may be modeled after a gate (system call) or process switch. In most existing architectures, an interrupt handler is a function that is invoked on an interrupt. The function executes as part of the interrupted process context or as part of a system-wide context. Although easy to implement, the function call does not isolate interrupt handlers, execute them at any level, or return from them to a different process.

The *WE* 32100 Microprocessor uses either the process switch or gate switch. In the process switch model, an interrupt (called a full interrupt in this case) causes an implicit process switch to a new process. In the gate switch model, an interrupt (called a quick interrupt in this case) causes an implicit gate to a handler function. When full interrupts are used, the processor interrupt mechanism meets the isolation and execution-level requirements because each interrupt handler is a separate process with its own execution stack. The processor tracks full-interrupt nesting in such a way that a full-interrupt handler at any priority level may preempt the original process, thus meeting the return requirement. With the quick-interrupt feature, interrupts can be handled as described above for most existing architectures.

For efficient operation, the implicit process switch on a full interrupt does the following:

- Minimizes the loading and saving of an interrupt handler's context
- Allocates only one stack to each interrupt-handler.

4.5.2 Interrupt Mechanism

There are three functions of the interrupt mechanism:

- Determining whether or not there will be an interrupt.
- Determining how an interrupt request will be acknowledged and what the interrupt-ID value is.
- Saving the old context and bringing in a new context.

The first part involves checking the $\overline{\text{NMINT}}$ and IPL[3—0] pins, and the IPL field of the PSW. The next part involves the $\overline{\text{NMINT}}$, $\overline{\text{AVEC}}$, IPL[3—0] and $\overline{\text{INTOPT}}$ pins, and an *interrupt acknowledge* or *auto-vector interrupt acknowledge* bus cycle. The final part involves the QIE field of the PSW and a quick-interrupt (gate-like) sequence or a full-interrupt (process-switch) sequence.

The following algorithm describes the interrupt behavior. The notation used is:

- I==1 if there is to be an interrupt
- ID is the value used as the interrupt-ID in the on-interrupt microsequence
- NMI, INTOPT, and AVEC represent the complements of the values of the nonmaskable interrupt ($\overline{\text{NMINT}}$), interrupt option ($\overline{\text{INTOPT}}$), and auto-vector ($\overline{\text{AVEC}}$) pins, respectively.

```
I=0;
if(NMI==1) {
        I=1;
        ID=0;
}
else if((requested_interrupt_level)>(PSW<IPL>)) {
        I=1;
        if(AVEC==1)
                ID=(INTOPT concatenated with interrupt request level);
        else ID=(value fetched in interrupt acknowledge cycle);
}
if(I==1) {
        call on-interrupt microsequence;
}
else {
        no interrupt;
}
```

An interrupt occurs if the priority level requested is greater than the priority level in the IPL field of the PSW. Thus, if PSW<IPL>==15, no interrupts will be acknowledged (except for nonmaskable interrupt).

After acknowledging an interrupt (full or quick as determined from Table 2-4), the processor performs its on-interrupt microsequence (an implicit process or gate switch). Its actions are similar to a call process (CALLPS) instruction for a full interrupt and a gate (GATE) instruction for a quick interrupt, but with a few differences.

When a full interrupt activates an interrupt-handler process, the interrupt handler starts from its initial state. However, unlike ordinary processes, this initial context consists of only the three registers and the stack bounds; general registers are not loaded for any process starting from an initial context.

A higher priority interrupt may interrupt the current interrupt-handler process. When this happens, its intermediate context is stored in the save area of the PCB, rather than the initial-context area. Thus, the interrupted interrupt handler can resume execution from that point later.

The I bit in the process PSW controls which starting point and context to use (see **4.4.1 Context Switching Strategy**).

To return from a full interrupt, an interrupt-handler process executes a return-to-process (RETPS) instruction. This process switch does not save the state of the exiting interrupt-handler process (see **4.4.3 Return-to-Process Instruction**).

When a quick interrupt activates an interrupt handler, the current PC and PSW values are stored on the execution stack. A simulated gate is then performed to load the PC and PSW registers with the initial information for the interrupt handler. A quick-interrupt gate does not perform any stack bounds check; therefore, quick interrupts should not occur in processes where the stack may be bad (e.g., a user process with a stack that is unreliable). Also, a quick-interrupt gate sets the PSW interrupt priority level (IPL) field to 15, thus disabling all interrupts except a nonmaskable interrupt.

Only a nonmaskable interrupt may interrupt the current quick-interrupt handler. When this happens, the PC and PSW values of the interrupted interrupt handler are stored on the execution stack and another simulated gate is performed. Thus, the interrupted interrupt handler can resume execution from its interrupted state.

To return from a quick interrupt, an interrupt handler should restore the IPL field in the PSW and then execute a return from gate (RETG) instruction (see **4.3.3 Return-From-Gate Instruction**).

Full-Interrupt Handler's PCB

Before an interrupt handler is activated, its PCBP points to the initial-context area of its PCB, which contains initial values for the PSW, PC, and SP. The IPL field in this PSW is usually set at least as high as the priority level of the device associated with the interrupt handler. (Interrupt-priority levels range from 0, the lowest, to 15, the highest, which indicates "no interrupts.") In addition, the I bit in this PSW should contain 1. If the interrupt handler wants to use the general registers, the PSW R bit should be 1.

If the new PSW has its I bit set when an interrupt handler is activated, the I bit in the PSW register is cleared and the PCBP register is adjusted to the saved-context area of the handler's PCB. The save area is used to store the handler's control registers if another interrupt occurs.

If the PSW's I bit is set, an interrupt-handler process always starts from the same initial state whenever it is initially activated because its initial-context values never change. However, after being interrupted, the saved-context area always reflects its state at the time of the interrupt. Thus, the restored interrupt handler starts from the appropriate intermediate state.

An interrupt handler's MMU map specification, if maintained in the PCB block-move areas, is used when loading an initial context or restoring an intermediate context. Therefore, the user must ensure that the operating system restores the map data to its initial state before a return-from-interrupt. This can be done by maintaining appropriate R bit values in the PCBs involved.

Interrupt Stack and ISP

The user must design the operating system to allocate memory space for one interrupt stack. This system data structure enables the processor to track the nesting of interrupt handlers and active processes and is never used as an execution stack.

The processor uses its interrupt stack pointer (ISP) register to access the interrupt stack. This privileged register always contains the address of the top of the stack. When it saves the current PCBP, a CALLPS or on-interrupt microsequence automatically increments ISP by four. A RETPS decrements ISP by four when it restores the PCBP. An attempt to write this register other than in kernel level causes a normal exception privileged register.

At any level of full-interrupt handling, the interrupt stack contains the PCBPs for all lower priority interrupt handlers that were interrupted while executing. The entry at the bottom of the stack is the PCBP for the process that was interrupted by the first interrupt handler (see Figure 4-5).

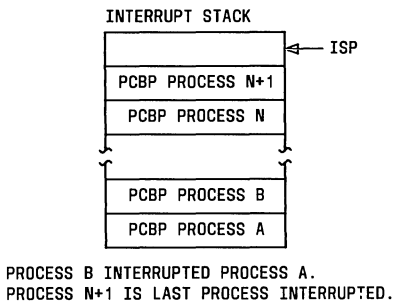
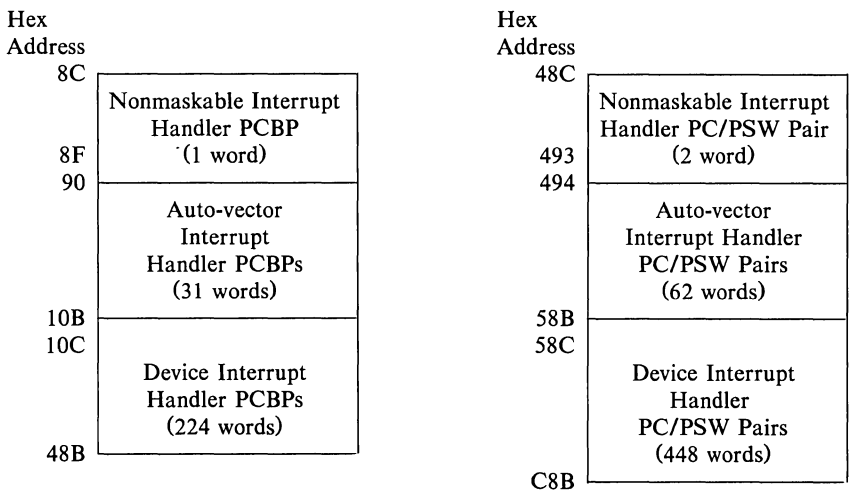


Figure 4-5. An Interrupt Stack

Because a return-from-process (RETPS) restores the process that was interrupted, the process at the bottom of the stack is eventually restored. However, any interrupt handler whose PCBP is on this stack may force a return to a different process. If any interrupt handler does this, be sure that it overwrites the normal-process PCBP at the bottom of this stack with the PCBP of the desired process.

Interrupt-Vector Table

The user must provide interrupt-vector tables for full and quick interrupts, depending on how interrupts are to be handled (process switches and/or gates). Figure 4-6 shows the memory locations where interrupt PCBPs and PC/PSW pairs must be stored. If the nonmaskable and auto-vector interrupts are not used, those locations can be used to store the PCBPs for device-interrupt handlers. The full-interrupt-vector table starts at location 140 (8C hex) to store the PCBP (up to 256 PCBPs) for each interrupt handler and the quick-interrupt-vector table starts at location 1164 (48C hex) to store PC/PSW pairs (up to 256 pairs) for each interrupt handler. Commonly, each device that requests an interrupt may require a different handling routine. The processor locates the appropriate interrupt handler by using an 8-bit code (interrupt-ID) as an offset into the vector tables. The code is used to form the address (140 + 4*interrupt-ID) to obtain the PCBP for a full-interrupt handler or the address (1164 + 8*interrupt-ID) to obtain the PC/PSW pair for a quick-interrupt handler.



A. Full-Interrupt Vector Table **B. Quick-Interrupt Vector Table**

Figure 4-6. Interrupt Vector Tables

4.5.3 On-Interrupt Microsequence

The on-interrupt microsequence is a sequence of actions built into the *WE* 32100 Microprocessor that responds to interrupts. The on-interrupt microsequence handles both full and quick interrupts. For full interrupts, the processor performs an implicit process switch. For quick interrupts, the processor performs a GATE-like PSW/PC switch. Here, Process A is the interrupted process and Process B is the interrupt handler. (See **4.4.2 Call Process Instruction** for descriptions of the XSWITCH functions.)

The microsequence performs the following sequential steps:

1. Writes the interrupt-ID to register tempa. If a memory exception occurs, the processor generates a stack exception (interrupt-ID fetch).
2. Forces kernel level on memory accesses.
3. Skips to step 12 if it is a quick interrupt (the PSW's QIE field is set to 1).
4 Performs steps 5 through 11 for a full interrupt.
5. Forms an index 140+4*tempa, which is written to tempa. This index is used to locate the PCBP of the appropriate interrupt handler.
6. Stores the Process A PCBP on the interrupt stack. If a memory exception occurs on this stack operation, the processor generates a reset exception (interrupt stack).
7. Calls XSWITCH_ONE() to store the Process A context in the saved-context area of its PCB and then writes 0, 0, 1 to the ISC, TM, and ET fields, respectively, of the saved PSW. If any of these operations causes a memory exception, the processor generates a process exception (old-PCB).
8. Calls XSWITCH_TWO() to load the Process B PCBP and new PC, PSW, and SP values from the initial-context area of its PCB. A memory exception on any XSWITCH_TWO operation causes a process exception (new-PCB). If it is set, the PSW I bit will be cleared and PCBP adjusted to the saved-context area of Process B PCB.
9. Writes 7, 0, 3 to the PSW's ISC, TM, and ET fields, respectively.
10. Calls XSWITCH_THREE() to make any necessary block moves. A memory exception here causes a process exception (new-PCB).
11. Releases kernel level on memory accesses. For full interrupts, this is the last step of the on-interrupt microsequence.
12. Resumes quick interrupt here.
13. Forms an index, 1164+tempa*8, which is written to tempa. This index is used to locate the PSW and PC of the appropriate interrupt handler.
14. Releases kernel level on memory accesses.

15. Pushes the PSW and PC of Process A onto the execution stack.
16. Forces kernel level on memory accesses.
17. Sets the PSW with value indexed by tempa, and PC with value indexed by 4+tempa. Some fields in the PSW are unchanged. Also, the IPL field is set to 15 to mask any subsequent interrupts. If a memory exception occurs, a normal exception (gate vector) is generated.
18. Releases kernel level on memory accesses. For quick interrupts this is the last step of the on-interrupt microsequence.

Process B (the interrupt handler) takes its priority level from the PSW that was just loaded and starts executing. Execution may be interrupted only by a higher priority interrupt (higher than the IPL value of the PSW).

4.5.4 Returning From an Interrupt

Full Interrupts

A full-interrupt handler may restore the interrupted process or may return to another process after servicing the interrupting device. To accomplish either process switch, the full-interrupt handler must contain a return-to-process (RETPS) instruction. Unlike the call process, RETPS does not save the exiting process (interrupt handler) context.

Note: If a full-interrupt handler is not to return to the process interrupted, the interrupt-handler routine must alter the interrupt stack before a RETPS instruction. The PCBP for the process returned to must replace the PCBP that was saved for the interrupted process.

The PCBP of the process to which the return-from-interrupt occurs is removed from the interrupt stack. The full context of the returning process is restored from its PCB, and any required map changes are made (block moves are performed).

Quick Interrupts

A quick-interrupt handler returns to the function that was interrupted (i.e., restores the PC and PSW registers with the values popped off the execution stack). To return from a quick-interrupt handler, the handler must execute a return-from-GATE (RETG) instruction. Also, before returning from a quick interrupt, the IPL field of the PSW should be set to the previous state of the interrupted process.

4.6 EXCEPTIONS

An *exception* is an error condition, other than an interrupt, that requires special processing for recovery. That is, an exception mechanism is needed to correct the error condition so

that normal processing can continue. Exceptions are caused by the following three types of events:

- Internal faults - error conditions detected by the processor during instruction execution. The fault handler for such events may restart the instruction that caused the fault.
- External faults - error conditions detected outside the processor and conveyed to it over its fault input. The processor recognizes the fault during instruction execution and the appropriate fault handler may then restart the execution.
- Traps - internal error conditions detected by the processor at the end of an instruction. After the trap is handled, execution may resume with the next instruction.

The exception mechanism for the *WE* 32100 Microprocessor is implemented through microsequences. Depending on the level of exception severity, the microprocessor responds with the appropriate microsequence to facilitate correction of the condition.

4.6.1 Levels of Exception Severity

The processor recognizes four levels of exception severity, with zero (0) as the highest level. It uses the ET (exception type) and ISC (internal state code) fields of the PSW to identify the severity and type of exception, respectively. Because all exception microsequences preserve the ET and ISC values in the current PSW, the incoming exception handler may use them. The ET value gives the class of exception and corresponds to its severity level, while ISC distinguishes among error conditions of the same class. During normal program execution, ET is 3 and ISC is 7. Table 4-4 identifies the severity levels, giving the ET value in decimal. The meaning of the ISC values for each exception severity level is identified later.

4.6.2 Exception Handler

On-stack, on-process, and on-reset exception microsequences do not use the ET and ISC values, but preserve them for an incoming exception handler. The on-normal exception microsequence uses them to locate the appropriate handling routine, as well as preserving them.

Table 4-4. Severity Levels for Exceptions

ET	Level	Processor Response
0	Reset	Executes on-reset microsequence; highest severity level
1	Process	Executes on-process exception microsequence
2	Stack	Executes on-stack exception microsequence
3	Normal	Executes on-normal exception microsequence; lowest severity level

The ET and ISC values help identify the task an exception handler must perform. What an exception handler should do with the ET and ISC values or how it should handle the error depends on the needs of the system. In general, if computation can continue, resumption of the process may be chosen. However, if an error is too serious for the original process to continue its computations, the exception handler should ask the scheduler to terminate the bad process.

The operating system designer must provide exception-vector tables. Figure 4-7 shows the addresses where the vector tables reside. All locations must be filled with either PCBPs or the address of the handling-routine table (for normal exceptions).

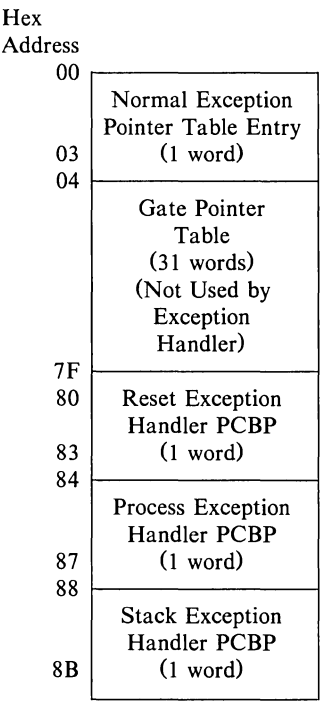


Figure 4-7. Exception-Vector Table

4.6.3 Exception Microsequences

The processor's microsequences enable it to execute an appropriate sequence of actions when it detects an exception. By design, an exception that occurs during one of these microsequences has a higher severity level. Such an exception, therefore, stops the current microsequence, and the processor starts performing a higher level microsequence. Thus, the processor can ripple up levels of exception severity.

Any exception during an on-reset sequence (the severest exception level) causes the processor to restart the on-reset sequence. Trying to recover from the exception, the processor goes into an infinite loop and consequently can recover from transient faults.

The sections that follow describe the error conditions for each class of exception and the response of the microsequence. When describing this response, Process A is the process that caused the exception and Process B is the exception handler. In general, a normal exception results in a simulated gate instruction, but a stack, process, or reset exception causes an implicit process switch. Descriptions of microsequences follow the operating system instructions at the end of this chapter.

Normal Exceptions

This group of exceptions includes most of those that occur in other microprocessor architectures. Table 4-5 identifies the ISC and the cause of each normal exception.

When a normal exception occurs, the processor executes the on-normal exception microsequence. After some set up operations, the microsequence enters the gate instruction at its second entry point (see **4.3.2 Gate Instruction**). Using the ISC code, this simulated GATE finds the appropriate exception-handler function and transfers control to it. Both the microsequence and the exception handler execute within the process that caused the error condition.

To locate the exception handler, GATE requires two implied operands that serve as indexes into the pointer table and the correct handling-routine table. (See **4.3.1 Gate Mechanism** for a description of these tables.) For GATE index1, the microsequence supplies the value of 0. For GATE index2, it uses the internal-state code (ISC) in the saved PSW, shifted three bits toward the most significant bit. This shifted ISC value forms an index into the handling-routine table. Thus, a normal exception results in a controlled transfer to the corresponding exception handler. On completion of the on-normal exception microsequence, the ISC, TM, and ET fields of the PSW presented to the exception handler will contain 7, 1, 3, respectively.

Because a normal-exception handler executes as part of Process A, it uses the same execution stack. After handling the error condition, a normal-exception handler must execute a return from gate instruction to restore control to Process A.

Table 4-5. Normal Exceptions (ET=3)		
ISC	**Exception**	**Cause**
0	Integer zero divide (Internal fault)	An attempt to divide by zero. This exception is always enabled. (Note 1)
1	Trace (Trap)	Normal response to the end of an instruction if the TE bit is set in the PSW.
2	Illegal opcode (Internal fault)	Use of an undefined opcode.
3	Reserved opcode (Internal fault)	Use of an opcode reserved for future implementation. This is also the normal response to the extended opcode (EXTOP) instruction.
4	Invalid descriptor (Internal fault)	Use of literal or immediate address mode for a destination operand; instruction's opcode requests the effective address of a literal, immediate, or register operand. (Note 1)
5	External memory (External fault)	A exception when accessing external memory.
6	Gate vector (External fault)	A memory exception when accessing the gate tables as part of a GATE.
7	Illegal level change (Internal fault)	An attempt to increase the current execution privilege level on a RETG.
8	Reserved data type (Internal fault)	Use of an operand type that is not defined for the expanded-operand type address mode. (Note 1)
9	Integer overflow (Internal fault)	An attempt to write data into a destination that is too small. This exception is enabled when the OE bit is set in the PSW. (Note 2)
10	Privileged opcode (Internal fault)	An attempt to execute an opcode defined for kernel level at a different execution level.
11–13	Unused	–
14	Breakpoint (Trap)	Normal response to a breakpoint trap (BPT) instruction.
15	Privileged register (Internal fault)	An attempt to write the ISP, PCBP, or PSW when not in kernel level. (Note 1)

Notes:
1. This exception sets the condition flags as if the instruction was successfully completed.
2. Before the overflow trap occurs, the processor may execute the next instruction after the one that caused the overflow.

Stack Exceptions

Table 4-6 lists the ISC and the cause of each stack exception. A stack-bound exception occurs when the stack-bound check fails on a system call (a gate instruction or on-normal exception microsequence). A stack fault occurs on an execution stack access to save the

current PC and PSW. An interrupt-ID-fetch exception occurs during the on-interrupt microsequence if an exception occurs during the acknowledge access.

On a stack fault, the memory exception occurs when SP is used as an operand. Thus, the processor first detects a normal exception and then detects the stack exception while executing the implicit GATE (system call). In effect, the processor automatically ripples up to a stack exception from a normal exception.

A stack exception occurs because Process A (the process at fault) cannot use its execution stack. As a result, a stack exception cannot be handled as part of Process A (unlike normal exceptions). Instead, the processor performs the on-stack exception microsequence, which performs a process switch and thus provides the exception handler with a new execution stack.

The interrupt-ID-fetch exception does not involve the stack, but it is treated as a stack exception since it is systemwide. Thus, no context information is lost.

The on-stack exception microsequence saves the Process A PCBP on the interrupt stack, stores the control registers in its PCB, and loads a new PCBP (for Process B) from location 136 (88 hex). Then it carries out an implicit process switch to the stack-exception handler, Process B. Although the microsequence does not use the ISC value, it preserves this value across the process switch. On completion of the microsequence, the ISC field in the PSW saved for Process A still contains the code for the stack exception, and the TM and ET fields contain 0 and 3, respectively. When Process B starts executing, the PSW's ISC, TM, and ET fields contain 7, 0, 3, respectively.

Because a stack-exception handler is implemented as a process, the user may want to prevent interrupts from entering the handler. Entry prevention is accomplished by raising the interrupt priority level (the IPL field of its PSW) to 15 and thus disabling all interrupts except a nonmaskable interrupt. Such a stack-exception handler should execute only a few instructions.

A stack-exception handler can correct a stack-bound or stack-fault problem by:

- Growing the stack of the process
- Bringing in a missing page of the stack (in demand-paging systems).

Table 4-6. Stack Exceptions (ET=2)

ISC	Exception	Cause
0	Stack bound (Internal fault)	An SP value outside the upper or lower stack bound on a system call.
1	Stack (External fault)	A memory exception when storing the PC or PSW on the execution stack during a system call.
3	Interrupt ID fetch (External fault)	A memory exception during the interrupt acknowledge access during an interrupt sequence.

Process Exceptions

A process exception is generated if the process receives a memory exception signal on a PCB access. The exception is local to Process A (the process that caused it) and implies a severe error condition. The ISC field of the Process A PSW is presented to the exception handler (Process B) and identifies the condition that caused the exception. Table 4-7 lists the ISC and the cause for each process exception.

When a process exception occurs, the processor executes its on-process exception microsequence, an implicit process switch. Because the error condition signifies that the Process A PCB cannot be accessed, its context cannot be saved. The microsequence stores the Process A PCBP on the interrupt stack and loads the Process B PCBP from location 132 (84 hex). Then it loads the Process B context, preserving the ISC value from the Process A PSW. When Process B begins executing, its PSW contains the code for the exception condition, and the TM and ET fields contain 0 and 3, respectively.

Because the processor could not save the Process A hardware context, Process B normally kills Process A. However, it can identify an old (good) process from its PCBP on the interrupt stack. If the exception is a new PCB exception, the Process A PCBP is at the top of the interrupt stack. If it is an old PCB exception and a process switch from a third process (Process C) had been made previously, then the Process C PCBP is the second element from the top of the stack. In either case, Process B could restart the last good process because its context was not lost.

Reset Exceptions

A reset exception implies an error condition in accessing critical system data and requires restarting of the system. On a reset exception, the processor acts as if an external reset occurred. The ISC field in the PSW of the current process identifies if the condition is an internal error or external request for a system reset. Table 4-8 lists the ISC and cause of the reset exceptions.

Table 4-7. Process Exceptions (ET = 1)

ISC	Exception	Cause
0	Old PCB (External fault)	A memory exception when accessing the PCB for the exiting process on a process switch.
1	Gate PCB (External fault)	A memory exception when accessing the PCB for a stack bounds check during a GATE.
4	New PCB (External fault)	A memory exception when accessing the PCB for the new process during a process switch.

On a reset exception, the processor performs an implicit process switch. It executes the On-Reset microsequence after first disabling the memory management unit. The microsequence picks up a new PCBP from physical address location 128 (80 hex) and loads the reset-handler process (Process B). When Process B begins executing, its PSW contains the code corresponding to the condition that caused the reset exception, and its TM and ET fields contain 0 and 3, respectively.

Process B should restart the system (i.e., reinitialize the system), possibly after checking the validity of system data.

Table 4-8. Reset Exceptions (ET=0)

ISC	Exception	Cause
0	Old PCB (External fault)	A memory exception when accessing the PCB of a process-exception handler.
1	System data (External fault)	A memory exception when accessing an interrupt vector or while processing an exception.
2	Interrupt stack (External fault)	A memory exception when accessing the interrupt stack while processing an exception.
3	External reset (External fault)	Normal response to an external (system) reset signal.
4	New PCB (External fault)	A memory exception when accessing the PCB of an exception-handler process.
6	Gate-vector (External fault)	A memory exception when accessing a gate table while processing a normal exception. (Here, the PSW ET field contains 0. If ET is 3, a gate-vector exception is treated as a normal exception because it occurred during a GATE instruction, rather than as part of the on-normal exception microsequence.)

4.7 MEMORY MANAGEMENT FOR VIRTUAL MEMORY SYSTEMS

When a virtual memory system is used for a *WE* 32100 Microprocessor based system, a memory management unit (MMU) is required. The main function of an MMU is to translate virtual addresses into physical addresses. The MMU has the additional responsibility of providing protection for the system memory space.

The virtual address space is divided into a number of sections by the MMU. Each section is in turn subdivided into segments. Segments may either be *contiguous* or *paged* and are mapped into physical address space by the MMU.

The *WE* 32101 Memory Management Unit (MMU) was developed to complement the *WE* 32100 Microprocessor for creation of a virtual memory system. This section describes the features of the MMU that are important for system design. A complete technical summary of the MMU is provided in the *WE* 32101 Memory Management Unit Data Sheet.

The *WE* 32101 Memory Management Unit divides the virtual address space into four sections and provides both contiguous and paged segments for the system. A contiguous segment can be as large as 128 Kbytes and a paged segment can contain up to sixty-four 2 Kbyte pages.

The MMU divides virtual addresses into three fields for contiguous segments and four fields for paged segments. A virtual address referencing a contiguous segment is divided into three fields: a section ID (SID) field, a segment select (SSL) field, and a segment offset (SOT) field. The SID field specifies the section of virtual address space, the SSL field specifies the segment within the section, and the SOT field specifies the byte within the segment. The format of these virtual addresses is shown on Figure 4-8.

For paged segments, the SOT field is subdivided into a page select (PSL) field and a page offset (POT) field. The PSL field specifies which page within the segment and the POT field specifies which byte within the page. The format of these virtual addresses is shown on Figure 4-9.

The MMU performs address translation using descriptors that contain the information necessary for segment and page mapping. The MMU has two types of descriptors: segment descriptors (SD) for mapping contiguous and paged segments and page descriptors (PD) for mapping pages within paged segments. An SD contains a segment base address that is added to an offset (from the virtual address SOT) to form the physical address. The PD contains a page base address that is concatenated with a page offset (from the virtual address POT) to form the physical address.

Other fields contained in SDs and PDs provide functions other than address translation. For example, the access fields in the SDs are used by the MMU to enforce protection of system memory. This field and other fields are described later in this section.

The SDs for each of the four sections of virtual memory are located in physical memory in segment descriptor tables (SDTs). There is one SDT associated with each section. The PDs for each paged segment are located in physical memory in page descriptor tables (PDTs), and there is one PDT associated with each paged segment. Contiguous segments are represented by an SDT entry, while paged segments are represented by both an SDT entry and an entire PDT (the SDT entry contains the physical base address of the PDT).

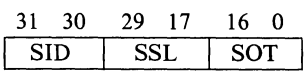


Figure 4-8. Virtual Address Fields For a Contiguous Segment

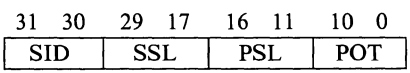


Figure 4-9. Virtual Address Fields For a Paged Segment

Figure 4-10 is a model showing how a virtual address is translated to a physical address for a contiguous segment. The SID field is used to find the base address of the required SDT. (The base address of the SDT for each section is stored in the MMU.) This address and the SSL field are combined to index an SD within the SDT. The starting physical address of the contiguous segment is contained in the indexed SD. This address is added to the SOT field to form the required physical address.

Figure 4-11 shows the paged segment model. This translation is identical to the contiguous segment address translation up to the point where the SD is indexed. For paged segments, the address in the SD is used as the base address of a PDT. This address is combined with the PSL field to index a PD. This PD contains the starting address of the paged segment that is concatenated with the POT field to form the required physical address.

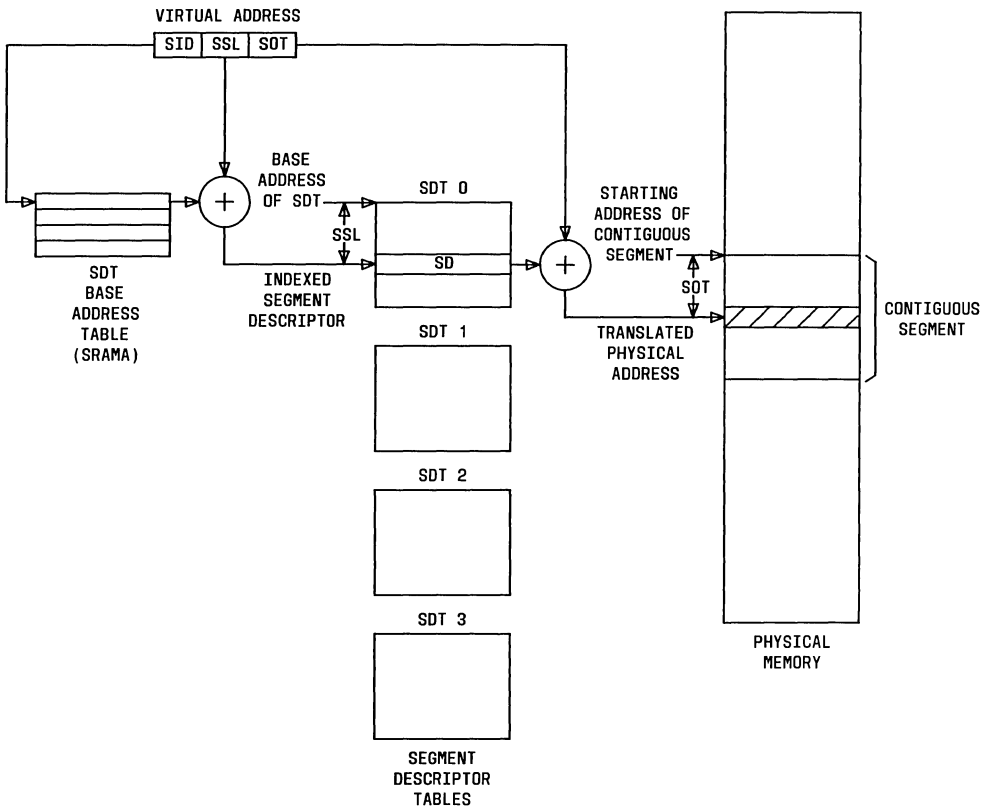


Figure 4-10. Virtual to Physical Translation for Contiguous Segments

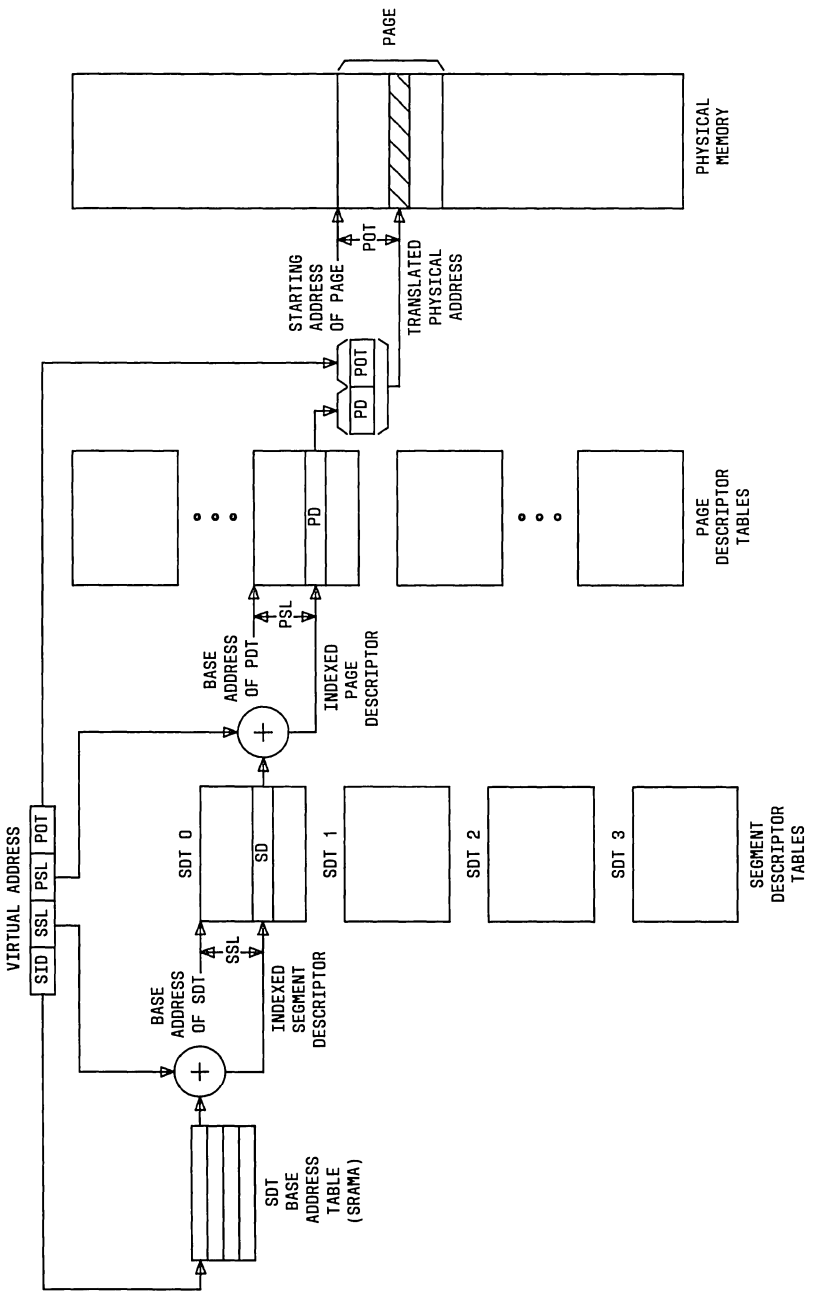


Figure 4-11. Virtual to Physical Translation for Paged Segments

4.7.1 Initializing the Memory Management Unit

The operating system is required to initialize the MMU. Typical MMU initialization consists of:

- Defining physical memory with segment descriptor tables and page descriptor tables for each process
- Writing SDT addresses and length into MMU section RAMs.

The operating system should also set up the block-move area of the process control block (PCB) for each process in the system. Block moves can be used to set the MMU section RAMs, if desired, when process switches occur. Setting the section RAMs causes the MMU to flush its caches.

Defining Virtual Memory

The operating system must define the way virtual memory is to be configured. In systems using an MMU this requires that segment and page descriptor tables be set up in physical memory. The way these tables are set up determines which segments in virtual memory are to be contiguous or paged and where the segments and pages reside in physical memory.

Peripheral Mode

The peripheral mode of the MMU is used by the operating system in several ways. One use is to initialize some of the internal elements of the MMU. The elements that require initialization are the section RAMs and the configuration register (CR). Section RAMs are loaded with the SDT's base addresses and length. The descriptor caches may be preloaded to avoid miss-processing (for a real-time process or other special case).

Other uses of the peripheral mode by the operating system include:

- Setting or clearing the configuration register referenced and/or modified bits
- Reading the fault code register (FLTCR) and fault address register (FLTAR) in order to handle MMU-generated exceptions
- Reading the cache contents in the case of serious exceptions (e.g., double-page-hit).

4.7.2 MMU Interactions

The MMU interacts with the operating system through address translation, miss-processing, exception detection, and other events. Once the MMU is initialized, it translates virtual addresses by using the SDs and PDs. It caches descriptors from the SDTs and PDTs to minimize translation time. The MMU handles the transfer of descriptors between its caches and physical memory during miss-processing without operating system intervention. The MMU also checks for violations (e.g., address or access) without operating system action. If violations occur, exceptions are issued and the operating system's exception handler can respond accordingly.

MMU Exceptions

Operating system action is required when the MMU signals to the CPU that an exception (external fault) has occurred. The MMU detects several exceptions that relate to errors (such as memory exceptions when the MMU does not correctly read an SDT or PDT) and places the corresponding code in the fault code register (FLTCR) and the fault address register (FLTAR).

Other exceptions signal that data is not present in physical memory. In these cases, the MMU tells the CPU that a required page or segment is not in physical memory and must be brought into physical memory. The operating system is responsible for these activities; it must do any I/O that is necessary and adjust the appropriate SDT and/or PDT values.

The MMU provides hardware support for operating system page- or segment-replacement algorithms by setting the R and M bits in the segment and page descriptors whenever a segment or page is referenced or modified. If the operating system periodically clears all of the R bits, for example, it can use the R bits to implement a variation of the least recently used (LRU) replacement algorithm. It could choose to replace segments or pages that still have their R bits clear when an exception occurs, reasoning that those segments or pages have been referenced less recently than the ones with the R bits set.

Flushing

The operating system occasionally alters the contents of the descriptor tables in memory. For example, it must do this to set and clear bits that indicate whether a page or segment is present whenever they are swapped in and out of physical memory. Any alteration of the table contents must be followed by some type of flushing of the MMU caches to prevent the chaos that would result if tables and caches contained conflicting information. If the operating system alters a table entry for one page or segment, it must flush the cache entry for that page or segment, if there is such a cache entry. If the operating system alters or deletes many entries in a table, it may be more efficient to flush an entire section than to flush several cache entries one at a time.

4.7.3 Efficient Mapping Strategies

The memory mapping defined by the operating system may have an enormous effect on the performance of the system. There are some basic rules for efficient mapping strategies. Large blocks that will remain in physical memory for long periods could be defined as contiguous segments so that few entries will be needed in the descriptor tables and descriptor caches. If physical memory is scarce, however, use of several large contiguous blocks could result in long waits to move the blocks in and out, thus wasting the physical memory where another large block cannot fit.

If only part of a segment need be in memory at a time, paged segments make more efficient use of memory.

4.7.4 Object Traps

Through object traps, the operating system can invoke a process or procedure whenever virtual addresses in a given segment are generated. The MMU can then save the virtual address that caused the trap. This facility can be used to make I/O devices or external processors appear as normal segments from the user-software point of view.

4.7.5 Indirect Segment Descriptors

Indirect segment descriptors provide a mechanism to create shared segments that may be easily swapped out. The only segment descriptor that has to be modified by the operating system when the shared segment is swapped or moved is the last one (i.e., the descriptor that directly references the segment data).

Indirect segment descriptors are useful for shared segments where different processes running at the same execution level are given different access permissions to the segment. The access permissions in the last descriptor are superseded by the access permissions in the first descriptor used in the reference.

Indirect segment descriptors can also be used to provide chains of descriptors so that the path to the last segment descriptor can be passed on from one process to another. This is similar to the passing of pointers in a programming language, except that here each process that owns a descriptor that others are linked to can rewrite that descriptor, thus breaking or redirecting the chain.

4.7.6 Using the Cacheable Bit

Cached segment and page descriptors each contain one cacheable bit (represented by $ for the MMU). Whenever a descriptor is used for translation, the MMU reflects the value of the $ bit in the cached descriptor through the cacheable (CABLE) output.

The $ bit in the segment descriptor is copied into the cached page descriptor during miss-processing so that (from the operating system designer's point of view) the $ bit values are associated with segments, not individual pages.

The MMU does not manipulate the $ bits and the CABLE output signal in any other way, so this facility can be used in any way desired by the system designer. As an example, one possible use (from which the name cacheable is derived) is to provide an interface to a cache memory other than the MMU's own descriptor caches. In this scenario, the cacheable bit is used to indicate the contents of the associated segment that are not cacheable.

4.7.7 Using the Page-Write Fault

The fault on write (W) bit in the MMU's page descriptors is checked during address translation after all other checks have been done. If the W bit is set and the access type is a write, a page-write fault occurs. This feature can lead to increased efficiency in the implementation of a *UNIX* System fork. The W bit could be set when the fork is invoked,

and then both the parent and child processes could continue to use those pages without having the MMU and operating system physically copy the shared pages until one of those pages is written. A write operation would cause a page-write fault, and the pages would be copied and the write bits reset. In this way the system copies pages only as necessary.

4.7.8 Access Protection

Access bits contained within segment descriptors specify the access permission (no access, execute-only, read/execute, and read/write) for each execution level (kernel, executive, supervisor, and user). These bits provide protection so that segments are accessed on the appropriate level. If an access permission is disallowed, an access exception occurs.

4.7.9 Using the Software Bits

Three software bits are contained in each segment and page descriptor. The MMU does not alter the value of these bits at any time. This allows the operating system designer to use these bits in any manner. For example, a software bit can be used to avoid allocating any stack space until a process actually needs it. This is done by assigning the software bit to signify that a page does not exist. Normally, a process start-up would create a (sometimes large) stack of zeros. The software bit could be used to avoid creating the stack until the user program references that page. Only then would the page-not-present fault cause the operating system to allocate the stack space. If the user program never references that page, the software bit saves memory for other processes.

4.8 OPERATING SYSTEM INSTRUCTIONS

The remainder of this chapter describes the operating system instructions (listed in Table 4-1) and the microsequences. Each description includes the assembler syntax, operation performed, effect of address modes on condition flags, exceptions generated, and an example.

Some operating system instructions and all microsequences call at least one XSWITCH function to do parts of the context switch. These functions, XSWITCH_ONE(), XSWITCH_TWO(), and XSWITCH_THREE(), are included among the microsequences.

4.8.1 Notation

Operations are described in C language where possible. In particular, the following notation is used where a C language operator or symbol did not exist:

*x	Word of register *x* contains the address of (a pointer to) the operand.
*x++	Use word or register *x* as a pointer to the operand; then increment *x* by 1, 2, or 4 for a byte, halfword, or word operation, respectively.

*--x	Decrement word or register x by 1, 2, or 4 for a byte, halfword, or word operation, respectively; then use *x* as a pointer to the operand.
interrupt_ID	An 8-bit value, generated on the interrupt acknowledge access cycle, identifies the interrupt vector to the process.
dst	Replace with destination operand.
src	Replace with source operand.
{operation}	Text between braces describes an operation in general terms.
R<a> = <x>	Replace field (or bits) *a* of word R with the value *x*.

Table 4-3 lists the symbols used to define the bits fields being altered in the PSW. See Tables 4-5 through 4-8 for the ISC values.

The following symbols are used to identify processor registers:

AP	Argument pointer, r10 (assembler syntax %ap)
FP	Frame pointer, r9 (assembler syntax %fp)
ISP	Interrupt stack pointer, r13 (assembler syntax %isp)
PC	Program counter, r15 (assembler syntax %pc)
PCBP	Program control block pointer, r14 (assembler syntax %pcbp)
PSW	Processor status word, r11 (assembler syntax %psw)
R*n*	Register *n*, r*n*,*n* = 0 to 8 (assembler syntax %r*n*)
SP	Stack pointer, r12 (assembler syntax %sp)

4.8.2 Privileged Instructions

These instructions are executed only when the process is in the kernel execution mode. Attempting to invoke them at a lower level causes a normal exception (privileged opcode).

Instruction	Mnemonic
Call process	CALLPS
Disable virtual pin and jump	DISVJMP
Enable virtual pin and jump	ENBVJMP
Interrupt acknowledge	INTACK
Return-to-process	RETPS
Wait	WAIT

The DISVJMP and ENBVJMP instructions disable or enable the processor's virtual address pin and then jump to an address. ENBVJMP enables an MMU, signalling that the processor is now supplying virtual addresses for translation. DISVJMP disables the MMU and only physical addresses are supplied. With an ENBVJMP instruction, a new (virtual) address is loaded into the PC; hence the jump. For DISVJMP, a physical address is loaded into the PC. The use of CALLPS and RETPS was previously discussed in **4.4.2 Call Process Instruction** and **4.4.3 Return-to-Process Instruction**, respectively. WAIT provides a processor-level execution halt that remains in effect until an interrupt occurs.

The following descriptions provide more detail about the instructions.

Call Process

Assembler Syntax CALLPS

Opcode 0x30AC

Description This instruction performs a process switch, saving the current process, pushing its PCBP onto the interrupt stack, and entering a new process. It:

- Saves the context (register contents) of the current process in the current PCB (if R bit of new process is set).
- Pushes the current PCBP value onto the interrupt stack.
- Puts the new PCBP value (from register **r0**) into the PCBP register.
- Sets the PSW, PC, and SP registers from the new PCB.
- Performs block moves (if any) for the new process (if R bit of PSW is set).
- Exits, going to the new process.

Operands **r0** is an implicit source operand (it should contain the PCBP of the new process).

Operation

```
if (!kernel-level)
   normal-exception (privileged-opcode)

/* put new PCBP into tempa */
tempa = r0

/* push old PCBP onto interrupt stack */
{force kernel level on memory accesses}
*ISP++ = PCBP
if(memory-exception)
      reset-exception(interrupt-stack)

/* Any memory exception in the first XSWITCH subroutine will cause
a process exception (old PCB). The address of the next instruction is
always PC + 2 */
PC = address of next instruction
/* set old PSW ISC/TM/ET to 0/0/1 respectively */
PSW<ISC> = 0
PSW<TM> = 0
PSW<ET> = 1
```

```
/* save current registers in current PCB */
XSWITCH_ONE()
/* XSWITCH_ONE() performs the following operations */
*(PCBP + 4) = PC
PSW<R> = *tempa<R>
*PCBP = PSW
*(PCBP + 8) = SP
if(PSW<R>) {
      *(PCBP + 20) = AP
      *(PCBP + 24) = FP
      *(PCBP + 28) = r0
            |
            |
            |
      *(PCBP + 60) = r8
      FP = PCBP + 52
}

/* Any memory exception in the following XSWITCH subroutines will
cause a process exception (new PCB). */

/* put new PCBP in PCBP register and get new PC, PSW, and SP. */
XSWITCH_TWO()
/* XSWITCH_TWO() performs the following operations */
PCBP = tempa
PSW = *PCBP /* PSW<R/ISC/TM/ET> unchanged here */
PSW<TM> = 0
PC = *(PCBP + 4)
SP = *(PCBP + 8)
if(PSW<I>) {
      PSW<I> = 0
      PCBP = PCBP + 12
}
if(PSW<CFD> == 0)
      {flush instruction cache}

/* set new PSW ISC/TM/ET to 7/0/3 respectively */
PSW<ISC> = 7
PSW<TM> = 0 /* avoid CALLPS trace trap */
PSW<ET> = 3
```

```
/* do block moves if PSW<R> is set (1) */
XSWITCH_THREE()
/* XSWITCH_THREE() performs the following operations */
if(PSW<R>) {
      r0 = PCBP + 64
      r2 = *r0++
      while(r2 != 0) {
            r1 = *r0++
            {execute MOVBLW instruction}
            r2 = *r0++
      }
      r0 = r0 + 4
}

{unforce kernel level on memory accesses}
{end of operation}
```

Address Modes None

Condition Flags Set by new PSW

Exceptions
normal exception (privileged opcode)
process exception (old PCB or new PCB)
reset exception (interrupt stack)

Example

```
{load new PCBP into r0}
CALLPS
```

Notes Opcode occupies 16 bits. The ISC/TM/ET fields of the PSW saved contain 0/0/1, respectively. These fields in the new process PSW contain 7/0/3, respectively.

Disable Virtual Pin and Jump

Assembler Syntax	DISVJMP
Opcode	0x3013
Description	This instruction changes the CPU to physical addressing mode (disables the MMU) and puts a new value in the PC (switching addressing modes usually makes the old PC value incorrect).
Operands	**r0** is an implicit source operand (it should contain the new physical PC value).
Operation	if(!kernel-level) normal-exception (privileged-opcode) {Reset virtual address pin ($\overline{\text{VAD}}$) to 1} PC = r0 {flush instruction cache}
Address Modes	None
Condition Flags	Unchanged
Exceptions	normal exception (privileged opcode)
Example	{load physical address of next instruction into r0} DISVJMP
Notes	Opcode occupies 16 bits.

Enable Virtual Pin and Jump

Assembler Syntax ENBVJMP

Opcode 0x300D

Description This instruction changes the CPU to virtual addressing mode (enables the MMU) and puts a new value in the PC (switching addressing modes usually makes the old PC value incorrect).

Operands **r0** is an implicit source operand (it should contain the new virtual PC value).

Operation

```
if(!kernel-level)
        normal-exception (privileged-opcode)
{Set virtual address pin (VAD) to 0}
PC = r0
{flush instruction cache}
```

Address Modes None

Condition Flags Unchanged

Exceptions normal exception (privileged opcode)

Example

```
{load virtual address of next instruction into r0}
ENBVJMP
```

Notes Opcode occupies 16 bits.

Interrupt Acknowledge

Assembler Syntax	INTACK *dst* interrupt acknowledge
Opcode	0x302F INTACK
Operation	under "interrupt acknowledge" status r0 ← (Interrupt – ID) << 2
Address Modes	None
Condition Flags	Unchanged
Exceptions	privileged-opcode exception
Examples	INTACK
Notes	This instruction is privileged. If NMINT==0 and AVEC==0, an "interrupt acknowledge" access is performed, fetching an 8-bit "interrupt-ID". This value is zero-extended to a word, shifted left by two bit positions, and stored in r0. If NMI==0, an "auto-vector-interrupt acknowledge" access is performed (with all 1s on the address bus) and 0 is stored in r0. If NMI==1 and AVEC==0, and "auto-vector-interrupt acknowledge" access is performed, and the "requesting level" (inverted and put on address bus and returned as "interrupt-ID") is indeterminate.

Return to Process

Assembler Syntax RETPS

Opcode 0x30C8

Description This instruction terminates the current process (its context is not saved) and returns to the process whose PCBP is on the top of the interrupt stack. It:

- Pops the saved (old) PCBP value from the interrupt stack.
- Puts the old PCBP value into the PCBP register.
- Sets the PSW, PC, and SP registers from the saved values in the old PCB.
- Performs block moves (if any) for the old process (if the R bit of the PSW is set).
- Puts the saved register values from the old PCB into the CPU registers (if the R bit in PSW is set).
- Exits, going to the old process.

Operands None

Operation

```
if (!kernel-level)
     normal-exception (privileged-opcode)

/* pop new PCBP from interrupt stack */
{force kernel level on memory accesses}
tempa = *--ISP
if(memory_exception)
      reset-exception(old-PCB)

/* Any memory exception in the following operation will cause a
process exception (old PCB).

Transfer R bit from new PSW to current PSW so block moves and
register restores will occur if needed. */
PSW<R> = *tempa<R>

/* Any memory exception in the following microsequence will cause a
process exception (new PCB).
```

```
Put new PCBP in PCBP register and get new PC, PSW, and SP. */
XSWITCH_TWO()
/* XSWITCH_TWO() performs the following operation */
PCBP = tempa
PSW = *PCBP /* PSW<R/ISC/TM/ET> unchanged here */
PSW<TM> = 0
PC = *(PCBP + 4)
SP = *(PCBP + 8)
if(PSW<I>) {
      PSW<I> = 0
      PCBP = PCBP + 12
}
if(PSW<CFD> == 0)
      {flush instruction cache}

/* set new PSW ISC/TM/ET to 7/0/3 respectively */
PSW<ISC> = 7
PSW<TM> = 0 /* prevent RETPS trace trap */
PSW<ET> = 3

/* do block moves, if R bit set */
XSWITCH_THREE()
/* XSWITCH_THREE() performs the following operation */
if(PSW<R>) {
      r0 = PCBP + 64
      r2 = *r0++
      while(r2 != 0) {
            r1 = *r0++
            {execute MOVBLW instruction}
            r2 = *r0++
      }
      r0 = r0 + 4
}

/* if R bit set, move saved register values from new PCB into CPU
registers. */
if(PSW<R>) {
      FP = *(PCBP + 24)
      r0 = *(PCBP + 28)
            |
            |
            |
      r8 = *(PCBP + 60)
      AP = *(PCBP + 20)
}
```

	{unforce kernel level on memory accesses} {end of operation}
Address Modes	None
Condition Flags	Set by new PSW
Exceptions	normal exception (privileged opcode) process exception (old PCB and new PCB) reset exception (interrupt stack)
Example	RETPS
Notes	Opcode occupies 16 bits. There is no check of the interrupt stack. Any exception in accessing this stack causes a reset.

Wait

Assembler Syntax	WAIT
Opcode	0x2F
Description	This instruction halts the CPU, stopping instruction, fetching, and execution until an interrupt or external reset occurs.
Operands	None
Operation	if(!kernel-level) normal-exception (privilege-opcode) {Halt CPU until an interrupt occurs}
Address Modes	None
Condition Flags	Unchanged
Exceptions	normal exception (privileged opcode)
Example	WAIT
Notes	Opcode occupies 8 bits.

4.8.3 Nonprivileged Instructions

These instructions are executed in any execution level:

Instruction	Mnemonic
Gate	GATE
Move Translated Word	MOVTRW
Return from Gate	RETG

GATE and RETG were discussed previously in **4.3.2 Gate Instruction** and **4.3.3 Return-from-Gate Instruction**, respectively.

MOVTRW tells an enabled MMU to intercept the virtual address sent by the processor, translate it, and return the physical address to the destination. If no MMU is enabled and the system treats the MT access as a read, then this instruction acts as a normal MOVW (i.e., the source is copied into the destination).

Gate

Assembler Syntax GATE

Opcode 0x3061

Description This instruction performs a system call, saving the current PSW and PC on the execution stack and using two levels of tables to obtain new PSW and PC values. It:

- Checks to make sure that the current stack pointer is within the stack bounds specified in the PCB. This is to insure that the routine called by the GATE instruction starts in a guaranteed *safe* stack area.
- Pushes a return address (PC) and the current value of the PSW on the execution stack. The return address insures that the GATE instruction can be used like a subroutine call. The PSW on the stack will be used by RETG to restore the CPU to the state it was in before the GATE function was invoked.
- Index1 is used as an offset into the first-level table, which starts at address 0. The word selected is the address of a second-level table.
- Index2 is used as an offset into the second-level table selected. It is added to the word read from the first-level table, to obtain the address of the PSW and PC entry in the second-level table. The first word of the entry selected is a new PSW to be used by the GATE-handling subroutine and the second word is the address (starting PC) of the gate routine.
- The PSW is replaced by the new PSW from the second-level table, with the old execution level field set appropriately and some other fields changed (see operation below).
- The PC is set to the address of the GATE-handling routine.
- GATE exits, going to the new PC.

Operands **r0** and **r1** are implicit source operands (they should contain byte offsets within first-level and second-level tables, respectively).

Operation /* When reading from the PCB in the following two operations, a memory exception causes a process exception (gate PCB).

```
Check SP against stack bounds in PCB. */
{force kernel level on memory accesses}
if(SP >= *(PCBP + 12))
        stack-exception (stack-bound)
if(SP >= *(PCBP + 16))
        stack-exception (stack-bound)
{unforce kernel level on memory accesses}

/* When writing to the stack in the following two operations, a memory
exception causes a stack exception (stack).

The address of the next instruction is always PC+2.

Save old PC and PSW on execution stack. */
*SP = address of next instruction
/* set PSW ISC/TM/ET to 1/0/2, respectively */
PSW<ISC> = 1
PSW<TM> = 0
PSW<ET> = 2
*(SP + 4) = PSW

/* mask index values and put in registers */
tempa = r0 & 0x7C    /* index1 */
tempb = r1 & 0x7FF8  /* index2 */

/* A memory exception from here to the end of the microsequence
causes a normal exception (gate vector).

Get new PC and PSW values from table. */
{force kernel level on memory accesses}
/* get pointer to second-level table */
tempa = *tempa
/* add offset within second-level table */
tempa = tempa + tempb

/* get new PSW from second-level table */
tempb = *tempa
/* set PM in new PSW to CM in old PSW */
tempb<PM> = PSW<CM>
/* new PSW same IPL/R values as old PSW */
tempb<IPL> = PSW<IPL>
tempb<R> = PSW<R>
/* set new PSW ISC/TM/ET to 7/1/3, respectively */
tempb<ISC> = 7
tempb<TM> = 1
tempb<ET> = 3
```

```
/* put new PC/PSW values into PC/PSW registers
   get new PC from second-level table */
PC = *(tempa + 4)
PSW = tempb

/* finish push of old PC and PSW */
SP = SP + 8

{unforce kernel level on memory accesses}
{end of operation}
```

Address Modes None

Condition Flags Set by new PSW

Exceptions
normal exception (gate vector)
stack exception (stack bound and stack)
process exception (gate PCB)
reset exception (gate vector)

Example GATE

Notes Opcode occupies 16 bits.

The values of **r0** and **r1** should be byte-valued offsets. The value of register **r0** must be a multiple of 4; and the value of **r1** must be a multiple of 8. These two registers are source operands only; GATE does not alter their contents.

Move Translated Word

Assembler Syntax MOVTRW *src,dst*

Opcode 0x0C

Description This instruction is intended for use with a memory mangement unit (MMU). An access using the address of the source operand and an MT access status is performed, and it is expected that the MMU will translate the address and return the corresponding physical address.

Operands *src* - contains virtual address to be translated
dst - contains the physical address after translation

Operation {under MT status}
dst = &src

Address Modes *src* - all modes except immediate, literal, or register
dst - all modes except immediate or literal

Condition Flags N = Bit 31 of word returned
Z = 1, if word returned == 0
V = 0
C = 0

Exceptions normal exception (invalid descriptor and external memory)

Example MOVTRW X,%r0

Notes Opcode occupies 8 bits.

When MOVTRW is executed in virtual mode with the *WE* 32101 Memory Management Unit present, the address is translated to the corresponding physical address. If there is no exception, the MMU returns the translated physical address, which is then stored at the destination. If there is an exception, the MMU notifies the CPU in the normal fashion.

When MOVTRW is executed in physical mode with the *WE* 32101 Memory Management Unit present, the MMU will behave as if a read operation in physical mode is taking place.

In systems without an MMU, some other device must respond to the MT access.

The source operand is an *address of* operand. The destination operand is of the type word. If *&src* is not a word address, a normal exception (external memory) will occur.

During an MOVTRW instruction, the status pins identify the memory access as being MT.

Return from Gate

Assembler Syntax RETG

Opcode 0x3045

Description This instruction can be used to return from a GATE, normal exception, or quick interrupt. The PC and PSW values to return to are popped from the execution stack, the current and new execution levels are compared to prevent a return to a higher execution level, and then the new values are put into the PC and PSW registers.

Operands None

Operation

```
/* get old PC/PSW values from execution stack */
tempa = *(SP - 4)
tempb = *(SP - 8)
if(memory-exception)
      stack-exception(stack)

/* compare execution levels to prevent return to a higher execution
level. */
if(tempa<CM> < PSW<CM>)
      normal-exception(illegal-level-change)

/* New PSW keeps same IPL/CFD/QIE/CD/R values as current
PSW. */
tempa<IPL> = PSW<IPL>
tempa<CFD> = PSW<CFD>
tempa<QIE> = PSW<QIE>
tempa<CD> = PSW<CD>
tempa<R> = PSW<R>
/* set new PSW ISC/TM/ET to 7/0/3, respectively */
tempa<ISC> = 7
tempa<TM> = 0 /* avoids RETG trace trap */
tempa<ET> = 3

/* put new PC/PSW values into PC/PSW registers */
PSW = tempa
PC = tempb

/* finish pop of old PC and PSW */
SP = SP - 8

{end of operation}
```

Address Modes	None
Condition Flags	Set by new PSW
Exceptions	normal exception (illegal level change) stack exception (stack)
Example	RETG
Notes	Opcode occupies 16 bits

4.8.4 Microsequences

The microsequences represent built-in microprocessor functions. These are executed automatically when the processor accepts an interrupt, generates an exception, or acknowledges a reset request. The XSWITCH functions are called by some operating system instructions and the microsequences.

On-Normal Exception

Description

A normal exception is caused by some action of the current process, such as execution of an illegal opcode, and it causes the CPU to perform the following GATE-like actions. This sequence is identical to that of GATE except that zero (instead of **r0**) is used as the offset into the first-level table (index1), and the ISC value (instead of **r1**) is used as the offset into the second-level table (index2).

A RETG instruction can be used to return from a normal exception.

Operation

```
/* When reading from the PCB in the following two operations, a
memory exception causes a process exception (gate PCB).

Check SP against stack bounds in PCB. */
{force kernel level on memory accesses}
if(SP < *(PCBP + 12))
      stack-exception(stack-bound)
if(SP >= *(PCBP + 16))
      stack-exception(stack-bound)
{unforce kernel level on memory accesses}

/* When writing to the stack in the following two operations, a memory
exception causes a stack exception (stack).

Save old PC and PSW on execution stack. */
*SP = PC
/* set PSW TM/ET to 0/3, respectively */
PSW<TM> = 0
PSW<ET> = 3  /* normal exception */
*(SP + 4) = PSW

/* set temp registers to GATE table index values */
tempa = 0
tempb = PSW<ISC> << 3

/* A memory exception from here to the end of the microsequence
causes a reset exception (gate vector).
```

```
Get new PC and PSW values from table. */
{force kernel level on memory accesses}
/* get pointer into second-level table */
tempa = *tempa
/* add offset within second-level table */
tempa = tempa + tempb
/* get new PSW from second-level table */
tempb = *tempa
/* set PM in new PSW */
tempb<PM> = PSW<CM>
/* set new PSW ISC/TM/ET to 7/1/3, respectively */
tempb<ISC> = 7
tempb<TM> = 1
tempb<ET> = 3

/* put new PC/PSW values into PC/PSW registers */
PC = *(tempa + 4)  /* get new PC */
PSW = tempb

/* finish push of old PC and PSW */
SP = SP + 8

{unforce kernel level on memory accesses}
{end of operation}
```

Condition Flags

Set by new PSW

Exceptions

stack exception (stack-bound and stack)
process exception (gate PCB)
reset exception (gate vector)

Notes

The value of the ISC field of the PSW is the identity of the normal exception. See Table 4-5 for a list of normal exceptions. The ISC field of the saved PSW contains this code.

Some exceptions set the condition flags as if the instruction that caused the exception was successfully completed.

On-Stack Exception

Description

A stack exception is caused by discovery of a stack-bound violation during a GATE or normal exception. Such an event causes the CPU to perform the following process switching action, similar to a CALLPS instruction except that the new PCBP is obtained from a fixed address instead of from **r0**.

A RETPS instruction can be used to return from the stack exception handler process.

Operation

```
/* Get new PCBP value from fixed address */
{force kernel level on memory accesses}
tempa = *136  /* 88 hex */
if(memory-exception)
      reset-exception(system-data)

/* push old PCBP onto interrupt stack :/
*ISP++ = PCBP
if(memory-exception)
      reset-exception(interrupt-stack)

/* Any memory exception in the first XSWITCH microsequence will
cause a process exception (old PCB). */
PSW<ET> = 2  /* stack exception */
PSW< ISC> = code for cause of exception
/* save current registers in current PCB */
XSWITCH_ONE()
/* XSWITCH_ONE performs the following operation */
*(PCBP + 4) = PC
PSW<R> = *tempa<R>
*PCBP = PSW
*(PCBP + 8) = SP
if(PSW<R>) {
      *(PCBP + 20) = AP
      *(PCBP + 24) = FP
      *(PCBP + 28) = r0
            |
            |
            |
      *(PCBP + 60) = r8
      FP = PCBP + 52
}
```

```
/* Any memory exception in the following XSWITCH
microsequence will cause a process exception (new PCB).

Put new PCBP value in PCBP register and get new PC, PSW, and
SP. */
XSWITCH_TWO()
/* XSWITCH_TWO performs the following operation */
PCBP = tempa
PSW = *PCBP /* PSW<R/ISC/TM/ET> unchanged here */
PSW<TM> = 0
PC = *(PCBP + 4)
SP = *(PCBP + 8)
if(PSW<I>) {
      PSW<I> = 0
      PCBP = PCBP + 12
}
if(PSW<CFD> == 0)
      {flush instruction cache}

/* set new PSW ISC/TM/ET to 7/0/3, respectively */
PSW<ISC> = 7
PSW<TM> = 0  /* prevent trace trap */
PSW<ET> = 3

{unforce kernel level on memory accesses}
{end of operation}
```

Condition Flags Set by the new PSW

Exceptions process exception (old PCB and new PCB)
reset exception (interrupt stack and system data)

Notes The ISC field of the saved PSW contains the code that caused the stack exception.

On-Process Exception

Description

A process exception is caused by a memory exception while accessing a PCB. Such an event causes the CPU to perform the following process switching action, similar to a CALLPS instruction except that there is no attempt to save the context of the current process (except for its PCBP value), and the new PCBP value is obtained from a fixed address instead of from **r0**.

There is no automatic way to return from a process exception because the exception is caused when there is a fatal error in the old process. The operating system is expected to choose some other process to invoke or return to.

Operation

```
/* Get new PCBP from fixed address. */
{force kernel level on memory accesses}
tempa = *132  /* 84 hex */
if(memory-exception)
        reset-exception(system-data)

/* push old PCBP onto interrupt stack */
*ISP++ = PCBP
if(memory-exception)
        reset-exception(interrupt-stack)

/* Any memory exception in the XSWITCH microsequence will cause
a reset exception (new PCB).

Put new PCBP value in PCBP register and get new PC, PSW, and SP.
*/
XSWITCH_TWO()
/* XSWITCH_TWO performs the following operation */
PCBP = tempa
PSW = *PCBP /* PSW<R/ISC/TM/ET> unchanged here */
PSW<TM> = 0
PC = *(PCBP + 4)
SP = *(PCBP + 8)
if(PSW<I>) {
        PSW<I> = 0
        PCBP = PCBP + 12
}
if(PSW<CFD> == 0)
        {flush instruction cache}
```

```
/* set new PSW TM/ET to 0/3, respectively */
PSW<TM> = 0  /* prevent trace trap */
PSW<ET> = 3

{unforce kernel level on memory accesses}
{end of operation}
```

Condition Flags Set by new PSW

Exceptions reset exception (system data, interrupt stack, and new PCB)

Notes The ISC field of the PSW presented to the exception handling process will contain the code corresponding to the condition that caused the process exception.

On-Reset Exception

Description

A reset exception is caused by an external reset request or by an exception while accessing the interrupt stack, the GATE tables, or the interrupt tables. Such an event causes the CPU to go to physical addressing mode, obtain a new PCBP value from a fixed address, and set the PSW, PC, and SP registers from values in the new PCB. No information from the current (old) context is saved because the CPU may be powering up for the first time or else the old software context was so damaged that it caused a reset exception.

Operation

```
{flush instruction cache}

if(external-reset)
      PSW<R> = 0

{force kernel level on memory accesses}

/* force physical mode */
{Set VAD pin to one}

/* get new PCBP from fixed address */
tempa = *128  /* 80 hex */
if(memory-exception)
      reset-exception(system-data)

/* Any memory exception in the XSWITCH microsequence will cause
a reset exception (new PCB).

Put new PCBP value in PCBP register and get new PC, PSW, and SP
values. */
XSWITCH_TWO()
/* XSWITCH_TWO performs the following operations */
PCBP = tempa
PSW = *PCBP /* PSW<R/ISC/TM/ET> unchanged here */
PSW<TM> = 0
PC = *(PCBP + 4)
SP = *(PCBP + 8)
if(PSW<I>) {
      PSW<I> = 0
      PCBP = PCBP + 12
}
if(PSW<CFD> == 0)
      {flush instruction cache}
```

```
/* set new PSW TM/ET to 0/3, respectively */
PSW<TM> = 0  /* prevent trace trap */
PSW<ET> = 3

{unforce kernel level on memory accesses}
{end of operation}
```

Condition Flags Set by new PSW

Exceptions reset exception (system data and new PCB)

Notes The ISC field of the PSW presented to the exception handling process will contain the code corresponding to the condition that caused the reset exception.

On-Interrupt

Description

An interrupt is triggered by a request from external hardware and causes the CPU to perform a process switch or a GATE-like action (depending on the value of PSW<QIE>).

For *full* (QIE==0) interrupts, the on-interrupt microsequence implements a process switch to the process represented by the PCBP value stored at location (140+(4*Interrupt-ID)), where *Interrupt-ID* is an 8-bit value fetched during an interrupt acknowledge access.

For *quick* (QIE==1) interrupts, the on-interrupt microsequence implements a GATE-like PSW/PC switch, pushing the old PSW and PC onto the execution stack and fetching new PSW and PC values from locations (1164+(8*Interrupt-ID)) and (1164+(8*Interrupt-ID)+4), respectively. However, quick interrupt does not perform any stack bounds check, so it should not be used with an untrusted user process, which may have a bad value in SP. Unlike GATE, quick interrupt does update the PSW<IPL> field to act.

If an interrupt request is granted and auto-vectoring is requested (via the AVEC pin), an auto-vector interrupt acknowledge cycle is performed and no *Interrupt-ID* is fetched. The complement of the value of the interrupt option pin concatenated with the priority level at which the interrupt was requested is used as the *Interrupt-ID*. That is, bits 0—3 of the ID correspond to the requested level, bit 4 corresponds to the interrupt option pin, and bits 5—7 are zeros.

If a nonmaskable interrupt request is received (via the NMINT pin), an auto-vector interrupt acknowledge cycle is performed (as if an autovector interrupt at level 0 was being acknowledged) and no *Interrupt-ID* is fetched. The value 0 is used as the ID.

Operation

```
{Get interrupt-ID value via interrupt acknowledge bus cycle}
tempa = interrupt-ID
if(memory-exception)
        stack-exception(interrupt-ID-fetch)

/* test for full or quick interrupt */
if(PSW<QIE>==1)
        goto QINT  /* quick interrupt */

/* it is a full interrupt */
```

```
/* get new PCBP from full interrupt table */
{force kernel level on memory accesses}
tempa = *(140 + tempa * 4)  /* 8C+tempa*4 hex */
if(memory-exception)
      reset-exception(system-data)

/* push old PCBP onto interrupt stack */

*ISP++ = PCBP
if(memory-exception)
      reset-exception(interrupt-stack)

/* Any memory exception in the first XSWITCH microsequence will
cause a process exception (old PCB).

Set old PSW ISC/TM/ET to 0/0/1, respectively. */
PSW<ISC> = 0
PSW<TM> = 0
PSW<ET> = 1

/* save current registers in current PCB */
XSWITCH_ONE()
/* XSWITCH_ONE performs the following operations */
*(PCBP + 4) = PC
PSW<R> = *tempa<R>
*PCBP = PSW
*(PCBP + 8) = SP
if(PSW<R>) {
      *(PCBP + 20) = AP
      *(PCBP + 24) = FP
      *(PCBP + 28) = r0
            |
            |
            |
      *(PCBP + 60) = r8
      FP = PCBP + 52
}

/* Any memory exception in the following XSWITCH microsequences
will cause a process exception (new PCB).
```

```
Put new PCBP value in PCBP register and get new PC, PSW, and
SP values. */
XSWITCH_TWO()
/* XSWITCH_TWO performs the following operations */
PCBP = tempa
PSW = *PCBP /* PSW<R/ISC/TM/ET> unchanged here */
PSW<TM> = 0
PC = *(PCBP + 4)
SP = *(PCBP + 8)
if(PSW<I>) {
      PSW<I> = 0
      PCBP = PCBP + 12
}
if(PSW<CFD> == 0)
      {flush instruction cache}

/* set new PSW ISC/TM/ET to 7/0/3, respectively */
PSW<ISC> = 7
PSW<TM> = 0  /* prevent trace trap */
PSW<ET> = 3

/* do block moves, if R-bit set (1) */
XSWITCH_THREE()
/* XSWITCH_THREE performs the following operations */
if(PSW<R>) {
      r0 = PCBP + 64
      r2 = *r0++
      while(r2 != 0) {
            r1 = *r0++
            {execute MOVBLW instruction}
            r2 = *r0++
      }
      r0 = r0 + 4
}

{unforce kernel level on memory accesses}
{end of operation}

QINT:  /* it is a quick interrupt */

/* Put address of new PC and PSW pair in quick interrupt table into
tempa. */
tempa = 1164 + tempa * 8  /* 48C+tempa*8 hex */

/* When writing to the execution stack in the following two operations,
a memory exception causes a stack exception (stack).
```

```
Save old PC and PSW on execution stack.
Note: No stack bounds check. */
*SP = PC  /* address of next instruction */
/* set PSW ISC/TM/ET to 1/0/2, respectively */
PSW<ISC> = 1
PSW<TM> = 0
PSW<ET> = 2
/* push PSW */
*(SP + 4) = PSW

/* A memory exception from here until the end of the microsequence
causes a normal exception (gate vector).

Get new PC and PSW values from table. */
{force kernel level on memory accesses}
tempb = *tempa
/* adjust previous execution level in new PSW */
tempb<PM> = PSW<CM>
/* set new PSW IPL to 15 */
tempb<IPL> = 15
/* new PSW ISC/TM/ET values same as old values */
tempb<ISC> = PSW<ISC>
tempb<TM> = PSW<TM>
tempb<ET> = PSW<ET>
/* put new PC/PSW values into PC/PSW registers */
PSW = tempb
PC = *(tempa + 4)

/* finish push of old PC and PSW */
SP = SP + 8

{unforce kernel level on memory accesses}
{end of operation}
```

Condition Flags Set by new PSW

Exceptions
normal exception (gate vector)
stack exception (stack and interrupt-ID fetch)
process exception (old PCB and new PCB)
reset exception (system data and interrupt stack)

Notes The interrupt-ID fetch is 8 bits, and is zero-extended to 32 bits in tempa.

XSWITCH Microsequences

Description

These microsequences implement context-switching. They are used, in various combinations, by the instructions CALLPS and RETPS and by the implicit microsequences On-Interrupt, On-Process, On-Stack, and On-Reset.

XSWITCH_ONE performs a context save, saving the current registers in the current PCB. XSWITCH_TWO performs a context switch, putting a new value in the PCBP register and reading the new PSW, SP, and PC values from the new PCB. XSWITCH_THREE performs block moves specified in the PCB.

The action taken when a memory exception is encountered in the XSWITCH microsequences is determined by the calling sequence.

Operation

```
/* Save current registers in current PCB.  One argument: tempa is
expected to contain new PCBP value. */

XSWITCH_ONE:

    /* save current PC in PCB */
    *(PCBP + 4) = PC

    /* copy R-bit from new PSW to current PSW */
    PSW<R> = *tempa<R>

    /* save current PSW and SP in PCB */
    *PCBP = PSW
    *(PCBP + 8) = SP

    /* if R-bit==1, save current r0-r8/FP/AP in PCB */
    if(PSW<R>) {
        *(PCBP + 20) = AP
        *(PCBP + 24) = FP
        *(PCBP + 28) = r0
              |
              |
              |
        *(PCBP + 60) = r8
        FP = PCBP + 52
    }

    return
```

```
/* Put new PCBP in PCBP register and get new PC, PSW, and SP.
One argument: tempa is expected to contain new PCBP value. */

XSWITCH_TWO:

    /* put new PCBP value into PCBP register */
    PCBP = tempa

    /* put new PSW, PC, and SP values from PCB into registers */
    PSW = *PCBP /* PSW<R/ISC/TM/ET> unchanged here */
    PSW<TM> = 0
    PC = *(PCBP + 4)
    SP = *(PCBP + 8)

    /* if I-bit==1, increment PCBP past initial context area */
    if(PSW<I>) {
        /* clear I-bit in PSW register */
        PSW<I> = 0
        /* increment PCBP past initial context area */
        PCBP = PCBP + 12
    }

    /* if cache flushing not disabled, flush cache */
    if(PSW<CFD> == 0)
        {flush instruction cache}

    return
```

```
/* do block moves, if PSW<R>==1 */

XSWITCH_THREE:

    if(PSW<R>) {

        /* get address of block0 size */
        r0 = PCBP + 64

        /* get block0 size */
        r2 = *r0++

        /* while block size != 0 */
        while(r2 != 0) {
            /* get destination start address */
            r1 = *r0++
            /* do one block copy */
            {execute MOVBLW instruction}
            /* get size of next block */
            r2 = *r0++
        }

        r0 = r0 + 4
    }

    return
```

Chapter 5

Software Generation Programs (Version SVR2.0)

CHAPTER 5. SOFTWARE GENERATION PROGRAMS

(VERSION SVR 2.0)

CONTENTS

CONTENTS

CONTENTS

5. INTRODUCTION TO THE SOFTWARE GENERATION PROGRAMS

The *WE* 321SG Software Generation Programs is a package of support software tools used to create and test programs for the *WE* 32100 Microprocessor. The SGP runs under the *UNIX* Operating System and uses many features of the *UNIX* System shell. The SGP makes possible high-level program coding and source-level testing of this code. This improves programming productivity by freeing programmers from hardware architectural details.

Since the SGP resides on a host *UNIX* Operating System, almost all user interaction with the *WE* 32100 Microprocessor goes through the host computer. The SGP imposes no convention on how the host computer ultimately communicates with the target *WE* 32100 Microprocessor.

The SGP frees programmers from the tedious task of machine-level coding and its pitfalls. The SGP provides symbolic programming on several levels and simplifies programming tasks by:

- Allowing programs to be portable across systems
- Making detailed knowledge of the *WE* 32100 Microprocessor architecture, I/O, and the operating system unnecessary.

Programs can be written in the lower-level assembly language, but need not be unless low-level data representations or low-level system functions must be accessed. Assembly language programming is used for applications requiring high levels of efficiency, or in cases where the higher-level language prevents access to data or to functions.

The C language is used as the high-level programming language. It contains a collection of control- and data-structuring facilities that greatly simplify programming tasks. Within the SGP, the C compiler **(m32cc)** converts C programs into assembly language programs that are ultimately translated into object files by the assembler **(m32as)**. The link editor **(m32ld)** collects and merges object files into executable load files. Each of these tools preserves all symbolic information necessary for meaningful symbolic testing at the C language source level. The SGP also provides a variety of utilities that read and manipulate object files.

Figure 5-1 shows the overall organization of the SGP. This organization conceptually parallels the C language support features of the *UNIX* Operating System.

Distinctive SGP Features

Distinctive features of the SGP tools are:

- All are designed to create and retain symbolic debugging information.
- A standard, common object file format is used.

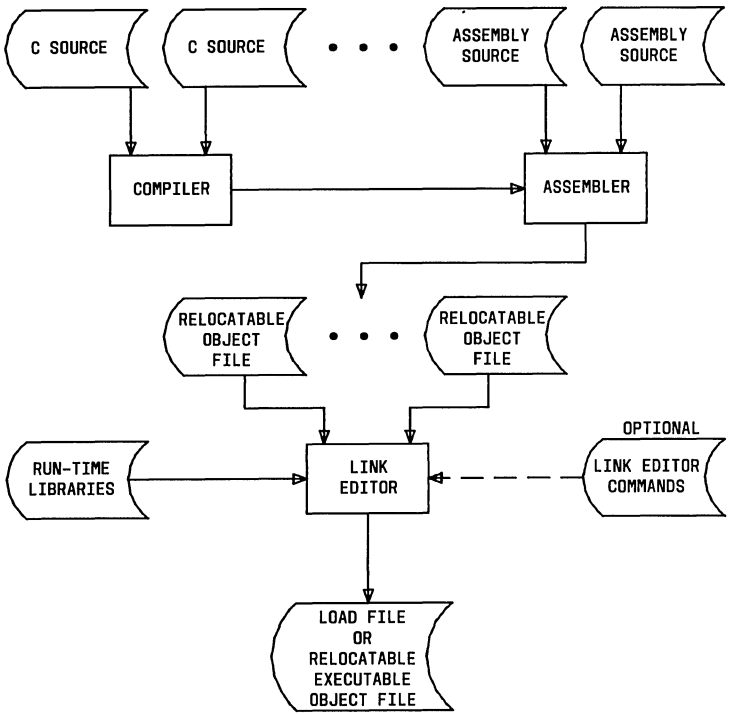


Figure 5-1. Major Steps in the SGP

The SGP emphasizes the generation and retention of symbolic debugging information. Table 5-1 lists the SGP tools described in this chapter. Also discussed are C language, assembly language, link-editor command language, and object file format.

Host Computers

The SGP runs under the *UNIX* Operating System, which in turn runs on a host computer. The host computer with the *UNIX* Operating System supports development of software for a target processor; in this case, the target is the *WE* 32100 Microprocessor. Other possible hosts for the *UNIX* Operating System and the SGP are:

- *AT&T* 3B20S Computer and *AT&T* 3B5 Computer
- Digital/Equipment Corporation *VAX* 11/780 Computer
- *IBM* 370 Computer running the "MAXI" version of the *UNIX* Operating System.

5.1 COMPILER AND THE C LANGUAGE

The C language is used for high-level programming and contains many control and structuring facilities that greatly simplify the task of algorithm construction. The C compiler (m32cc) converts C programs into assembly-language programs. Outputs the current on-line manual page for the compiler.

Table 5-1. SGP Tools

Tool	Description
m32ar	Combines several files into one archive file.
m32convert	Converts object and archive files into common object file format
m32cc	C Compiler
m32as	Assembler
m32ld	Link Editor
m32conv	Converts object files from one host machine format to another host machine format.
m32cprs	Compresses object files by removing duplicate structure and union descriptors.
m32dis	Disassembles object files to allow assembly-level debugging.
m32dump	Dumps selected parts of the named object files.
m32list	Produces a C language source list with line numbers that specify where breakpoints can be inserted.
m32lorder	Generates an ordered listing of object files suitable for link editing in one pass, as done by **m32ld**.
m32nm	Prints the symbol table for an object file.
m32size	Reports the number of bytes of text, uninitialized data, and initialized data (and their sum) included in an object file.
m32strip	Reduces file storage overhead by removing symbolic debugging information from an object file.

5.1.1 Compiler

The command for the compiler is **m32cc**. Prior to using the compiler, a file containing C source code is created using the *UNIX* Text Editor. The name of the file must end with the last two characters **.c** (e.g., **file1.c**). The command line

m32cc *options file.c*

is then entered to invoke the compiler on the C source file *file.c* with the appropriate *options* selected from Table 5-2. The compilation process creates an absolute binary file named **m32a.out** that reflects the contents of *file.c* and any referenced (user-supplied) library routines. The file, **m32a.out**, can then be executed on the target system.

options control the steps in the compilation process. When none of the controlling options are used, the **m32cc** compiler automatically calls the **m32as** assembler, and the **m32ld** link editor (see Figure 5-1).

The **m32cc** compiler also accepts input file names ending with the last two characters **.s**. The **.s** signifies a source file in assembly language. The **m32cc** compiler passes this type of file directly to **m32as**.

The **m32cc** compiler, based on a portable C compiler, translates C source files into assembly code. Whenever the command **m32cc** is used, the C preprocessor is called. The preprocessor performs file inclusion and macro substitution. The preprocessor is always invoked by **m32cc** and not called directly by the programmer. The expanded files are translated from C language to assembly code. Then, unless the appropriate flags are set, **m32cc** calls the assembler, optimizer, and the link editor to produce an executable file.

Compiler Options

All options recognized by the **m32cc** command are listed in Table 5-2 and on the manual page in **5.6 SGP MANUAL PAGES**. The following provides additional information for those options not completely described in Table 5-2.

By using appropriate options, compilation can be terminated early to produce one of several intermediate translations such as relocatable object files (**−c** option), assembly source expansions for C code (**−S** option), or the output of the preprocessor (**−P** option). Generally, the intermediate files may be saved and later resubmitted to **m32cc** with other files or libraries included as necessary.

When compiling C source files, the most common practice is to use the **−c** option to save relocatable files. Subsequent changes to one file do not require that the others be recompiled. A separate call to **m32cc** without the **−c** option creates the linked, executable **m32a.out** file. A relocatable object file created under the **−c** option is named by replacing **.c** with **.o** of the source filename.

The **−W** option provides the mechanism to specify options for each step that is normally invoked from the **m32cc** command line. These steps are:

- Preprocessing
- Compiler
- Optimization
- Assembly
- Link editing

The most common example of the use of the **−W** option is

−Wa,*−m*

which passes the **−m** option to the assembler. Specifying **−W***l*,*−m* passes the **−m** option to the link editor.

When the **−P** option is used, the compilation process stops after completing only preprocessing, with output left in *file.i*. This file is unsuitable for subsequent processing by **m32cc**.

The **−O** option decreases the size and increases the execution speed of programs by moving, modifying, merging, and deleting code. However, line numbers used for symbolic debugging may be transposed when the optimizer is used.

Table 5-2. m32cc Command Line Options

Option	Argument	Description
−c	None	Suppress the link-editing phase of compilation and force an object file to be produced even if only one file is compiled.
−g	None	Produce symbolic debugging information.
−p	None	Reserved for invoking a profiler.
−D	*identifier*[=*constant*]	Define the external symbol *identifier* to the preprocessor and give it the value *constant* (if specified). See Note.
−E	None	Suppress compilation and loading; i.e., invoke only preprocessor and direct the output to the standard output.
−I	*directory*	Change the algorithm that searches for **#include** files whose names do not begin with "/" to look in the named *directory* before looking in the directories on the standard list. Thus, **#include** files whose names are enclosed in " " are first searched for in the directory of the file being compiled, then in directories named by the **−I** options, and last in directories on the standard list. For **#include** files whose names are enclosed in <>, the directory of the *file* argument is not searched. See Note.
−O	None	Invoke an object code optimizer.
−P	None	Same as the **−E** option except output is directed to corresponding files suffixed **.i**.
−S	None	Compile the named C language programs, and leave the assembly-language output on corresponding files suffixed **.s**.
−U	*identifier*	Undefine the named *identifier* to the preprocessor. See Note.
−V	None	Print versions of **m32cc** and tools it invokes.
−y	*limit*	Allow user to set limit on the percent growth per file from in-line expansion. Values for *limit* are: **u**, allows unlimited growth; integer ≥ 0, allows indicated percent growth; **s**, suppresses in-line procedure expansion.
−W	*c,arg1*[,*arg2*...]	Pass along the argument(s) *argi* to pass *c*, where *c* is one of **[p02al]**, indicating preprocessor, compiler, or link editor, respectively. See Note.

Note: Argument is appended to option with no embedded blanks.

If an *asm* instruction is encountered under the **−O** option, the optimizer suppresses optimization of any function containing an *asm*.

The **−g** option produces information for a symbolic debugger. The SGP does not currently support a symbolic debugger, but one may be available as part of an application.

Register Usage

With the **−O** option, the compiler and optimizer provide automatic global register allocation on a procedural basis. Automatic allocation tries to move quantities to the scratch registers that are not saved/restored during procedure call/return. Also, it attempts to move quantities that cannot be placed in scratch registers into saved registers, if there is a net payoff. The movement into registers is impeded by constraints that restrict the registers' quantities. First, quantities that can be addressed in more that one way cannot be safely placed in registers. Second, scratch registers are changed by calls to procedures or move block instructions. Third, the number of registers is finite. And fourth, there is an overhead for using saved registers.

For most uses, the details of register usage or assignment are not needed by programmers. Registers can be accessed through an assembler escape, although this practice is not recommended. Registers have the following usage in the compiler:

- r0—r2 Scratch registers
- r3—r8 Saved register variables
- ap Argument Pointer
- sp Stack Pointer
- fp Frame Pointer

Six saved register variables are allowed by **m32cc** and are assigned to r8—r3 in descending order. If more than six registers are declared in a C source program, the compiler silently assigns stack space instead.

Register 0 (r0) holds the return value from a function call. Registers 0 (r0) and 1 (r1) hold the return value from a call to a double precision floating point function. For a function returning a structure, r2 passes the address in which the returned structure value should be stored. Function calls are assumed to require all scratch registers.

5.1.2 C Language

The C language used by the *WE* 32100 Microprocessor has features to accommodate both systems and general-purpose programming. The version of C language used is the one described in **The C Programming Language** by B. W. Kernighan and D. M. Ritchie (Prentice-Hall, 1978), except that it includes recent enhancements to C language. This section describes the extensions to C language not covered in Kernighan and Ritchie's book.

With the *WE* 32100 Microprocessor, C language data types map in the natural way for a 32-bit processor. That is, **char** maps to the processor type byte (8 bits), **int** and **long** map to word (32 bits), and **short** to halfword (16 bits). The compiler also accepts floating point data types. Codes for these data types assemble to opcodes which are illegal on the *WE* 32100 Microprocessor. Applications can trap on these opcodes and provide emulation of floating point operations.

C language leaves identification of the assembler escape keyword (*asm*) to the designer. The *asm* has been implemented for **m32cc** with the syntax:

asm ("assembly instruction").

For example,

asm ("movw &0,% r0")

loads register r0 with a 0. The assembly language instruction within the quotation marks is transmitted unchanged to the assembler.

The C language enhancements recognized by **m32cc** are:

- Flexnames
- Enumarations
- Structure Assignments
- Structure-valued arguments
- Functions returning structure values
- Nonunique structure member names

A detailed discussion of each enhancement follows. These details are not required by many programmers, but are included to completely describe the C language used by the processor.

Flexnames

Flexnames allow the use of arbitrary length variable names. The restriction of eight significant characters for C language variable names is removed. To allow names of arbitrary length, a string table was added to the object file, and the symbol table was modified to support the string table (see **5.4 OBJECT FILE FORMAT).**

Enumerations

Enumerations are unique data types with named constants. These partly replace the use of *#define* constants and offer the advantage of scoped constant names and strong type checking in the use of such names. Enumerations are analogous to the scalar types of the Pascal language.

To the type-specifiers listed in Section 8.2 of **The C Programming Language** by Kernighan and Ritchie, add:

enum-specifier

with the syntax

enum-specifier:

enum {*enum–list*}
enum *identifier* {*enum-list*}
enum *identifier*

enum-list:

enumerator
enum-list , enumerator

enumerator:

identifier
identifier = constant-expression.

The role of the identifier in the enum-specifier is entirely analogous to that of the structure-tag in a struct-specifier; it names a particular enumeration. For example,

```
enum color { red, green, yellow, blue };
...
enum color *cp, col;
...
col = yellow;
cp = & col;
...
if( *cp == green)...
```

makes color the enumeration-tag of a type describing various colors and then declares cp as a pointer to an object of that type and col as an object of that type.

The identifiers in the enum-list are declared as constants and may appear whenever constants are required. If no enumerators with "=" appear, then the values of the constants begin at zero and increase by one, as the declaration is read from left to right. An enumerator with "=" gives the associated identifier the value indicated; subsequent identifiers continue the progression from the assigned value. For example,

```
enum interrupt{
        halt = 0,
        bad_instr = 01001,
        mem_fault,
        div_zero = 02001,
        overflow,
        underflow
} icode;
...
if( (int)icode & 02000 )/* arithmetic fault */
...
```

illustrates specific value specification. In particular, the symbol overflow has the internal value 02002.

All enumeration constants must be distinct. Unlike structure members, enumeration constants are drawn from the same set as ordinary identifiers.

Objects of a given enumeration type are regarded as having a type distinct from objects of all other types. The compiler maps enumerations into the **int** storage class.

Structure Assignment

Structure assignment was added to the C language to simplify the transferring of the value of one structure instance to another, and to allow functions to return aggregate values. Structure assignment permits more efficient use of the processor and also improves source program readability.

Structures may be assigned as a unit, passed as arguments to functions, or returned by functions. All structure operands taking part in these operations must be of the same type. The following example demonstrates the new structure assignment features:

```
struct clock {
                    int hour, minute, second;
};
struct date {
                    int year, month, day;
                    struct clock time;
};
struct clock now={13,2,36};
extern struct date spring();
struct date today, tomorrow;

struct date nextday( day ) struct date day; {
                    struct date tempday;
                    ...
                    return tempday;
}

main() {
                    today = spring();
                    tomorrow = nextday( today );
                    tomorrow.time = now;
                    ...
}
```

Nonunique Structure Member Names

The current standard C language allows more flexibility in the reuse of structure member and structure field names than the original. The C language now permits reuse of structure member or field names. The exception is that a particular name may not be used for two distinct members within the same structure. This enhancement will, in one case, preclude the use of a type of reference to structure members that was permitted in older versions of the C language. This obscure case, where upward compatibility has not been maintained, is explained in detail. Nonunique member names permit more natural structure and union member naming conventions; which result in stronger and more efficient type checking of both structure and union member references.

Former Member Name Restrictions. Prior to this change, there were only two ways in which structure member names could be reused.

1. Names of members of two distinct structures could be identical only if those names represented the same member type and offset. For example, the name xyz is used in both of the following two structures:

```
struct s1 {
                long abc;
                char xyz;
                int def;
};
struct s2 {
                long rst;
                char xyz;
                short jkl;
};
```

   With such a construction, the structure member name xyz could be referenced from any structure variable of type s1 or s2, or any pointer to these types without ambiguity.

2. Member names could be reused within a new name scoping (block) level. In the following code section, the member name f_one is reused:

```
struct outer {
                int f_zero:2,f_one:4,f_two:10;
                struct outer *next;
};
funct() {
                struct inner {
                  int f_one, g_one, h_one;
                };
...
}
```

When member names are redeclared at different block levels, the innermost declaration serves to block the outer declarations of the same name within the inner scope. In a structure of the type outer, the four-bit field f_one could *not* be referenced within the function funct. This restriction would hold even for structures that were explicitly declared to be of type outer.

New Flexibility for Member Names. The language change for structure member names allows the reuse or redeclaration of structure member or field names with only a single restriction:

> A particular name may not be used for two distinct members within the same structure. A name may, however, be reused within nested structures.

Due to this change, type-checking is performed more strongly for structures and unions. A structure (or union) member is referred to as unique if it is declared only once, or if all its declarations conform to the requirements of Case 1 above. If a uniquely named member is mentioned in a structure reference where it is not a member of the structure, a warning diagnostic is issued. This allows old C language programs that violate these new rules to continue to compile. However, if a member that is not uniquely named is used in a structure reference while it is not a member of the structure, a fatal diagnostic is issued.

The case in which upward compatibility is not maintained involves structure member name declarations of Case 2 above.

```
struct x {
                int a,b;
                }x_obj;
main() {
                int *ip;
                struct{
                  int b,a;
                }y_obj;
                ...ip→a...
                ...y_obj.a ...
                ...x_obj.a ...
}
```

In the example above, prior to the language change, each of the references ip→a, y_obj.a, and x_obj.a was considered legitimate, and an offset of one word for the integer referenced by "a" was used. With nonunique structure members, the integer referenced by "a" in x_obj.a would have an offset of zero bytes from the address of x_obj. The portable C compiler used by **m32cc** considers such a reference to be a user error and issues a fatal diagnostic for ip→a.

Complete Structure and Union Member Reference Qualifications

Complete qualifications are now required for structure and union member references in the C language because ambiguities can arise with incomplete qualifications and nonunique structure member names. Incomplete qualifications are flagged with fatal error messages.

In earlier C compilers, a reference to a structure or union member could be abbreviated in some cases. When an abbreviation was used, a structure or union reference became a chain of member references (also called qualifications).

Qualifications were prefixed either by a structure or union proper or by a pointer to a structure or union. Because each qualification implied the addition of an offset within an address computation, it had been possible to omit those qualifications that had an offset

of zero. Zero offsets occur in the first member of a structure and in all members of unions. With the following two declarations:

```
struct xx {
                    struct yy {
                      int y1; char y2;
                    } ym;
                    ...
} *xp;
union u {
                    struct a {
                      int a1,a2,a3;
                    } mema;
                    struct b {
                      char b1,b2,b3;
                    } memb;
} *up;
```

the following references were allowed:

```
xp→y2   /* same as */   xp→ym.y2
up→b2   /* same as */   up→memb.b2
```

Of the references in the previous example, only the following structure and union member references are now legitimate:

```
xp→ym.y2
up→memb.b2
```

Nonunique Tag Names Allowed

Declared types of structure, union, and enumeration can be named by tag names that appear after the keywords *enum*, *struct*, and *union*, as shown in the following examples:

```
typedef enum bool {false,true} bool;
struct list *head;
union cell {unsigned word; char byte[2];};
```

Previous implementations of the C language required that all union and structure tag names be distinct from member names. The recent enhancements remove this restriction. As a result, four name pools now exist:

- **#define** macro names
- Structure, union, and enumeration tag names
- Structure and union members (which may be nonunique)
- All other names, including typedef, array, structure instance, and variable names; and enumeration constant names.

Vertical Tab Character Literal

The vertical tab character literal has been added to the C language. The character VT (octal 013 in ASCII) can now be represented as \v in addition to \013. This character may also be used within character string literals (e.g.: Upper left\t\t\t\v\vLower right\n). Vertical tab is now included in the definition of *white space* and therefore can be used to delimit tokens in a source file.

In-Line Procedure Expansion

With the **−O** option, the optimizer provides performance enhancements by expanding small procedures in-line to reduce discontinuities and the number of saves and restores executed. The optimizer expands a call to a procedure only if, after global register allocation, the procedure has no local variables and no saved registers, and if the call appears in the same file in which the procedure appears. When a procedure is expanded in-line this fact is noted in the object file symbol table (see the description of **Auxiliary Table Entries** in **5.4.7 Symbol Table**).

Procedures are expanded to only one level (i.e., calls within expanded procedures are not expanded). When the optimizer expands a procedure, it leaves the original copy in place, but strips the *call*, *save*, *restore*, and *ret* instructions from the copies expanded in-line. A procedure always appears once as a complete routine, but may appear many times as an in-line expansion. Arguments to an in-line copy are placed on the stack and referenced with the frame pointer of the calling routine. If a nested call is expanded, the frame pointer offsets for the in-line copy's argument references are corrected for the presence of other arguments on the stack.

The optimizer controls the amount of code growth resulting from the expansion by limiting the percentage code growth per file. It does this by controlling the number of calls expanded. The limit on the percent growth per file can be set by the user and a default value can be set at SGP build time.

5.2 ASSEMBLER AND ASSEMBLY LANGUAGE

This section describes the *WE* 32100 Microprocessor assembler (**m32as**) and assembly language. Most applications of the processor involve programming in a C language environment only. However, some applications may require assembly language programming for speed or access to functions not accessible at the C level. Short, frequently executed routines, such as the ones needed to handle I/O, interrupts, and device drivers are most likely written in assembly language.

The assembler constructs an object file from an assembly language source file. The object file is relocatable and may include an extensive symbol table for symbolic debugging. This relocatable object file is in "common" object file format and can be linked to other such files using the **m32ld** link editor.

The assembler translates operation code mnemonics and operands into the target machine bit pattern representing the particular instructions. The **m32as** assembler attempts to optimize its output, thus reducing the number of machine cycles required for a given task.

This optimization improves program speed. The assembler resolves local text labels, identifies global text symbols defined in the input files, and identifies symbols referenced but not defined.

The assembly language is made up of the *WE* 32100 Microprocessor instruction set, assembler directives, and a machine-independent instruction set. The machine-independent instructions are mapped into one or more *WE* 32100 Microprocessor instructions. The processor instruction set contains special-purpose I/O and system instructions local to the processor and a syntax for the variety of addressing modes that can be used to encode operand references. The assembler directives, called pseudo-operations (or *pseudo-ops*) permit description of high-level symbols and their types and storage classes, thus facilitating symbolic testing. Source line numbers can also be described. Other assembler directives can set location counters to allow flexibility in coding multiple sections in a single file.

5.2.1 Assembler

The assembler is normally called by the **m32cc** command rather than directly by the user. It has no flags of its own when called by **m32cc**, although it can be invoked directly with the command line

m32as *options filename*

where *options* are chosen from Table 5-3.

Table 5-3. m32as Command Line Options

Option	Argument	Description
−m	None	Invoke the **m4** macro processor.
−n	None	Turn off long/short address optimization.
−o	*objfile*	Place the assembled output in *objfile*.
−V	None	Print the version of the assembler being run on standard error.

The input assembly language program is read from *filename*, and the output is written to an output object file. Unlike **m32cc**, only one file at a time may be input to **m32as**. If the output file name is not specified by the **−o** option, the output name is created from *filename* using the following algorithm:

- If *filename* ends with the two characters **.s**, the output name is created by replacing these last two characters with **.o**.
- If *filename* does not end in **.s** and is no more than twelve characters in length, the output name is created by appending **.o** to *filename*.
- If *filename* does not end with **.s** and has more than twelve characters, the output name is created by appending **.o** to the first twelve characters of *filename*. (File names on the *UNIX* Operating System can be no longer than fourteen characters).

Usage of the assembler options entails a few potential pitfalls. If the **−n** option is not used, address optimization is invoked. The **.align** assembler directive is not guaranteed to work in a **.text** section when optimization is performed. Therefore, aligned constants should not be defined in the **.text** section. See **5.2.2 Assembler Directives** for a more detailed description of **.align**.

When the assembler is implicitly run by using **m32cc**, there are no key or reserved words. However, when the assembler is run explicitly, macro processing may be invoked. In this case, M4 keywords and predefined macros must not be used as symbols (variables, functions, labels) in the input file, since the macro processors cannot distinguish assembler symbols from macros. If macro expansion is not required, this problem cannot occur.

Assembled Files

The output of the assembler is an object file that has the format described in **5.4 Object File Format**. Each assembled file contains three sections: **.text**, **.data**, and **.bss**. Each section begins at an address that is a multiple of four and consists of a contiguous sequence of bytes. The **.text** section is used for the executable statements, the **.data** section is used for the initialized variables, and the **.bss** section is used for the uninitialized variables. Every statement in the input assembly language that produces code or data generates it into one of these sections.

The assembler maintains three location counters for each assembled file, one for each of the program sections. The initial value of each counter is set to zero. When an assignment is made to the corresponding program section, the assembler increments the appropriate location counter. On its final pass, the assembler concatenates the three sections for each file in the order **.text**, **.data**, and **.bss** and sets each location counter to the correct starting address. That is, the text origin is set to zero; the data origin is set to the location that follows the **.text** section; and the **.bss** origin is set to the location that follows the data entry. Figure 5-2 shows these starting memory locations.

Because the assembler produces relocatable code, modular program development is possible and is encouraged.

Diagnostics

Many different errors may occur when using the assembler. Nearly as many error messages are possible. The error messages are intended to be self-explanatory.

The most common error occurs when the input file cannot be read. The assembly then terminates with the message "Can't open *filename*". If assembly errors are detected in the input file, the following information is written to standard error: the input file name, the line number where the error occurred in the assembly code, and possibly a descriptive message for the problem. If the input file is produced by the C compiler, the line number in the C source program that generated the erroneous code is written on standard error.

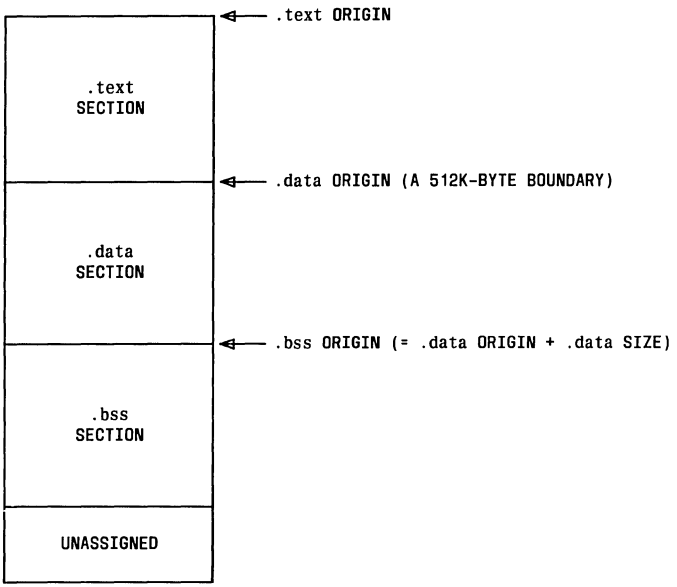


Figure 5-2. Mapping Program Sections

Macro Processing Facilities

Macro processors enhance programming languages by making them more readable, or by tailoring them to specific applications. The basic facility provided by any macro processor is replacement of text by other text. The **#define** statement in the C language performs a function for the compiler analogous to the function performed for the assembler by the macro processor.

When the **−m** option of **m32as** is specified, the M4 processor is invoked. The M4 macro processor provides a collection of about thirty-two built-in (default) macros; in addition, the user can define new macros using the M4 **define** function. As part of the programming environment provided by the SGP, many interfacing macros have been predefined. That is, the **define** function of M4 has already been used to establish several macros that interface assembly language routines with C code.

The M4 processor operates by copying its input to its output. As the input is read, each alphanumeric token (i.e., string of letters and digits) is checked. If the token matches the name of a macro, the name of the macro is replaced by the defining text, and the resulting string is pushed back onto the input and rescanned. In M4, built-ins and user-defined macros work exactly the same way, except that some of the built-in macros have side effects on the state of the process. Macros may be called with arguments, in which case the arguments are collected and substituted into the right places in the defining text before that text is rescanned.

Use of the M4 helps facilitate symbolic debugging when assembly code is used by tailoring the input file to look as though it came from the compiler. When an assembly language program uses the provided M4 macros, symbol table information can be generated, as well as the prologue and epilogue pseudo-code sequences that the compiler normally provides. The assembly language programming example demonstrates the prologue and epilogue sequences. (See **5.2.4 Programming Example**.)

Interface Macros

A set of predefined macros is provided to enable assembly language function linkages to C code to be specified independently from the details of the calling sequence. The macros therefore not only make programming easier; they also provide some insulation from any changes to the calling sequence that may occur. It must be emphasized, however, that while these macros make assembly language programming easier, they do not change the fact that, whenever possible, C language code should be used. Assembly code, no matter how well designed, is more difficult to write and debug than C code. In addition, assembly language routines do not necessarily perform a given task faster than high-level programs.

When the **−m** option is used, M4 preprocesses all input assembly language source files. The macros described below are made available as part of this preprocessing step. The M4 processor operates on both assembly language source files and on intermediate assembly language files generated by the compiler for C source files (i.e., **.c** files) that contain *asm* assembler escapes.

Note: When using **m32as**, the **−m** option can be specified on the command line. When using **m32cc**, the **−Wa, −m** option must be specified to access the macro package.

Function Interface Macros. The M4 macro package uses a functional notation for macros with arguments. Function interface macros should appear alone on a line with the arguments enclosed in parentheses and separated by commas. Additional white space (blanks and tabs) is ignored. Macros without arguments should appear in the assembly text just as if they were normal assembly language expressions.

C_PROLOGUE(*name*[,*nregs*])

This macro generates the standard C function prologue that finishes saving the caller's environment on the stack and sets up a new stack frame for use by the called routine. The operand *name* is the function name in the C source code; e.g., **prefix** in the example shown in **5.2.4 Programming Example**. The *name* must be a valid C language identifier.

The optional argument *nregs* gives the number of C language register variables that are saved by **C_PROLOGUE** (default is six registers). The assembly language function may use the saved registers for any purpose. Register variable arguments and stack arguments

are not available to **C_PROLOGUE**. Another predefined macro, **_RESULT**, names the register that must be loaded with any value to be returned to the calling function.

C_RETURN*(nregs)*

This macro generates the standard function return sequence. It restores the caller's environment and executes a branch to the return address that was saved with the environment on the stack at the time of the call. The number of registers to be restored is given by *nregs* and should be the same as that specified in **C_PROLOGUE**. The default is six.

C_CALL*(func*[*,arg1,...,arg5*])

This macro generates a call to the C language function *func*. The operand *func* must be a valid function name for either another normal assembly routine or a C source function that has become known by link editing. Up to five arguments can be passed with **C_CALL**. The arguments can be any valid operands to the assembler **pushw** instruction. Note that the function arguments are passed through without change (except for macro expansion). In the assembler language syntax, a variable name or constant operand is normally treated as if addressing a word in memory. The ampersand (&) can be used to show that the address itself is wanted. Thus, to use a specific value as an argument, an ampersand is used with the value. For example, the value 3 would be designated by &3. An argument that is to be the value stored at some address is indicated by giving the address with no ampersand. For instance, to obtain the contents at address x, designate the letter x. If the address itself is to be used as the value, write the value as an ampersand address; e.g., designate address x by &x.

A_PROLOGUE*(name)*

This macro operates the same as **C_PROLOGUE**, but does not allow any registers to be saved.

A_EPILOGUE*(name)*

This macro generates the symbolic code indicating the end of a function. Programmers must still write the actual return instructions before the **A_EPILOGUE** macro call; e.g., **RESTORE** and **RET**. Lines 30 through 33 in the example shown in **5.2.4 Programming Example** show the code generated by the **A_EPILOGUE** macro.

The macros that begin with **C** were written to connect assembly language segments to C language programs. However, they can also be used to connect two assembly language segments. In this use, the macros provide symbol table definitions, beginning and ending statements, and a save instruction for the new segment.

If only the symbol table definition and the beginning and end statements are needed, the **A_PROLOGUE A_EPILOGUE** pair should be used. The pair does not contain a save command, and its use requires explicit coding of save and return instructions.

Scratch Register Macros. The C compiler uses three scratch registers to store temporary results of expression computations. When the compiler processes a function call, it guarantees that no current values in the scratch registers will be needed after the call (by storing the values in temporary locations on the stack if necessary). Therefore, each function is free to use the scratch registers in any way and does not have to save or restore them. The macros **_SCR1**, **_SCR2**, and **_SCR3** expand to the register numbers of the scratch registers and may be used freely inside a normal assembly language routine. Note that **_SRC1** names the same register as **_RESULT**. Register **_SCR1** has special meaning during the call and return sequence, but is available for general use inside the called function.

Stack Frame Macros. Stack frame macros start with an underscore (_) and provide access to the current stack frame environment. The argument macros **_1STARG**, **_2NDARG**, **_3RDARG**, **_4THARG**, and **_5THARG** reference the first through fifth arguments to the function (via memory address), respectively. The macros **_1STREG**, **_2NDREG**, **_3RDREG**, **_4THREG**, **_5THREG**, and **_6THREG** reference the six general purpose registers, r8 through r3, respectively. The macro **_RESULT** references the register (typically r0) used by the C compiler to contain the value returned from a function.

If these macros are used in a normal assembly language routine (for example, one that uses **C_PROLOGUE** and **C_RETURN**), they refer to the stack frame set up by **C_PROLOGUE**. Note that **C_PROLOGUE** does not allocate any automatic storage.

The C stack frame can also be accessed directly by the stack pointer register (SP, r12), the frame pointer register (FP, r9), and the argument pointer register (AP, r10). The function interface and stack frame macros track any changes in the calling sequence. If the SP, FP, or AP registers are used to get closer to the stack frame layout, code will no longer be insulated from the details of the stack frame, and may have to be rewritten later.

Restrictions. In effect, the argument and register macros independently follow the same algorithm used by the C compiler to allocate storage. Because there is no way for the macro processor to know about the real environment of the assembly function or calling function, the following restrictions must be considered when using these macros:

- The use of argument and register macros is inherently machine-dependent; the macros cannot be recognized by processors not based on the assembler.
- All arguments, up to and including the last argument referenced by the macros, must be **ints** or pointers. These macros do not deal with **char**, **short**, or **struct** arguments. Functions that return structures require a more complicated calling sequence that is not handled by this macro package.
- For assembly language routines, any copying of arguments into registers must be done explicitly by the assembly code.
- Macro usage is not checked during the compiling and assembling of programs. Therefore, an assembly language routine that incorrectly changes the value of FP will cause run-time errors rather than compile-time errors.

Using Predefined Macros

A normal assembly language routine is called from a C source program just like any other function. The routine can have arguments passed to it, and it establishes its own environment on the stack. The file containing the assembly language source must have a name ending in **.s**. The **.s** tells the compiler (**m32cc**) to skip compilation and send the source directly to the assembler.

Examples. In the following example, a function named bump adds one to its argument and returns that result.

```
C_PROLOGUE(bump)
        movw    _1STARG,%_RESULT
        addw2   &1,%_RESULT
C_RETURN
```

If *bump* were called by the following C language routine

```
main()
{
        int i = 3;
        int j;
        j = bump(i);
}
```

then j would have the value 4.

The next example gets two pointers as arguments and swaps the values pointed to:

```
C_PROLOGUE(swap)
   movw _1STARG,%_1STREG      #1st arg is a pointer
   movw 0(%_1STREG),%_SCR1    #get value pointed to
   movw _2NDARG,%_2NDREG      #2nd arg is also a pointer
   movw 0(%_2NDREG),%_SCR2    #get its value
   movw %_SCR2,0(%_1STREG)    #store 2nd args value
   movw %_SCR1,0(%_2NDREG)    #store 1st args value
C_RETURN
```

Suppose *swap* was called by the following program:

```
main()
{
        int i = 3;
        int j = -4;
        swap(&i,&j);
}
```

Then i would get the value -4 and j would get the value 3. A C language function to accomplish the same task is

```
swap(i,j)
int *i,*j;
{
        register int temp;
        temp = *i;
        *i = *j;
        *j = temp;
}
```

In the final example, assembly function *chkster* checks to see whether, after stepping the first character, a text string has a common prefix with the string "abcdef" is defined using the function *prefix*. (See **5.2.4 Programming Example.**) This is a contrived example that has no place in real code, but is presented to demonstrate how a C language function is called with the **C_CALL** macro.

```
C_PROLOGUE(chkstr)
        addw3 &1,_1STARG,%_SCR1 #skip first character
        C_CALL(prefix, & string, %_SCR1)
C_RETURN
        .data
string:
        .byte 0x61,0x62,0x63,0x64,0x65,0x66,0x0
```

Note that the address of the format string must be passed to *prefix* and that the null byte terminating the string must be explicitly coded. Also note that unlike some implementations, the **m32cc** compiler does not prepend an underscore before global names. Thus *prefix* is used in assembly code, not *_prefix*.

M4 Reserved Words

Detailed discussion of the M4 processor can be found in the *UNIX* **System User's Manual.** A list of the M4 reserved words is:

changecom	ifdef	shift
changequote	ifelse	sinclude
decr	include	substr
define	incr	syscmd
defn	index	sysval
divert	len	traceoff

divnum	m4exit	traceon
dnl	m4wrap	translit
dumpdef	maketemp	undefine
errprint	popdef	undivert
eval	pushdef	

5.2.2 Assembly Language

This section describes the *WE* 32100 Microprocessor assembly language syntax and semantics. The basic actions of evaluation, assignment, and control of evaluation order are specified by statements. Statements are either machine instructions, assembler directives, or macro instructions.

The data types supported by the assembly language are byte, halfword, word, and bit field. A byte is an 8-bit quantity; a halfword is a 16-bit quantity; a word is a 32-bit quantity; and a bit field is a sequence of 1 to 32 bits.

The instruction set provides that bytes, halfwords, and words can be interpreted as either signed or unsigned quantities for arithmetic or logical operations. The processor does not generate any fault internally in the event of word or halfword data specified at improper addresses. The memory subsystem must generate a memory fault if such a fault is to be provided.

Detailed information on the instruction set, if needed, may be found in Chapter 3.

Statements

An assembly language program consists of a sequence of lines of code. Each line consists of a sequence of characters terminated by the new-line character (\n). Each line may contain one or more statements. If several statements appear on a line, they must be separated by semicolons (;). Each statement must be one of the following:

- Assembler Directive - a statement that is a command to the assembler. It consists of a pseudo-operation code followed by zero or more operands.
- Machine Instruction - a mnemonic representation of an executable machine instruction. It consists of an operation code followed by zero or more operands.
- Machine Independent Instruction - a statement that maps into one or more executable machine instructions.
- Empty - a statement that contains only spaces and tabs. It signifies nothing to the assembler, but is often used to enhance program readability.

Operation codes are separated from their operands by at least one space or tab. Operands and arguments are separated by commas. Unless otherwise stated, any other use of space and tab characters is optional. White space characters may be used freely to improve readability.

Each statement may be modified by one or more of the following:

- A label may be placed on any statement. The label consists of a *symbol* that begins in the first character position of a statement (i.e., it must begin IMMEDIATELY after a new-line character or semicolon) and is followed by a colon. *Symbols* are described in detail in the following section. An unlabeled statement MUST have a space, tab, or pound sign (#) in the first character position.
- A comment may be inserted at the end of any statement by preceding the comment with a pound sign. The assembler will ignore the pound sign and all characters following it up to the first new-line character. A new statement begins with the first character after the new-line character.

There are no limits on the number of characters in a statement or on the number of statements on a line. Multi-line comments are made by inserting a pound sign as the first nonwhite-space character of each line.

An example showing the four parts of assembly language statements follows. The first statement shows an assembler directive. The second statement is empty and was inserted to provide a visual break between directive and machine-instruction sections. The last two statements are machine independent instructions.

```
Label   Mnemonic   Operand      Comment
        .globl     prefix

main:   save       &.R1         #begin the function
        addw2      &.F1,%sp
```

These statements are taken from the example in **5.2.4 Programming Example**.

Symbols

Symbols are tokens recognized by the assembler. They always have a value and type, either specified explicitly by an assignment statement (see **5.2.2 Assembler Directives**) or determined from the context. Value and type are described in detail in this section. A symbol name consists of a string of the characters a—z, A—Z, 0—9, underscore (_), and period (.). Names may not begin with a digit. Because embedded blanks are not permitted in symbols, the underscore is generally used in place of a blank to make an identifier more readable.

Symbols are primarily used as labels. Four examples of symbols are:

```
Rtn_Nam5    abc    DEF    xyz.QQQ.
```

The assembler does not put symbols beginning with . (read as 'dot') into the object file symbol table. Exceptions to this rule are **.text**, **.data**, and **.bss**; these symbols are used for relocation.

The following symbols are reserved for use by the assembler:

1. . This symbol (read as dot) is used as the location counter while assembling a program. Whenever actual code is generated by the assembler, the value of this symbol is increased by the size of the generated code. Hence, this symbol effectively represents the address of the code being generated. Depending on the section for which code is being generated, dot may be of type TEXT, DATA, or BSS. Null data can be generated by pseudo-op assignment to this symbol.
2. **.text** This symbol has type TEXT and is used to label the beginning of the **.text** section for the program being assembled.
3. **.data** This symbol has type DATA and is used to label the beginning of the **.data** section for the program being assembled.
4. **.bss** This symbol has type BSS and is used to label the beginning of the **.bss** section for the program being assembled.

Values and Types. Values are represented in the assembler by signed 32-bit 2's complement numbers. Every value is an instance of one of the following types:

TEXT — A TEXT value is one that is defined relative to the beginning of the **.text** section. Whenever the **.text** section is relocated forward (backward) by N bytes, the number N will be added to (subtracted from) every value of type TEXT. The most common example of a TEXT value is a label appearing in the **.text** section.

DATA — A DATA value is one that is defined relative to the beginning of the **.data** section. Whenever the **.data** section is relocated forward (backward) by N bytes, the number N will be added to (subtracted from) every value of type DATA. The most common example of a DATA value is a label appearing in the **.data** section.

BSS — A BSS value is one that is defined relative to the beginning of the **.bss** section. Whenever the **.bss** section is relocated forward (backward) by N bytes, the number N will be added to (subtracted from) every value of type BSS.

UNDEFINED — An UNDEFINED value is one whose type has not yet been determined. The UNDEFINED value may be a reference to a symbol whose definition has not been encountered yet (i.e., a forward reference) or a reference to a symbol that is assumed to be defined in a program other than the one currently being assembled (i.e., an external reference).

ABSOLUTE — An ABSOLUTE value is one that will not change as a result of relocating any section of the program being assembled. Constants described in the following section have absolute type.

In addition, any of the above types may be given the attribute EXTERNAL. For values of the types ABSOLUTE, TEXT, DATA, and BSS; the attribute EXTERNAL indicates that a value defined in the program currently being assembled will be made available to other programs. For values of type UNDEFINED, EXTERNAL means that the value is referenced in the program currently being assembled, but is defined in some other program.

Assigning Values and Types to Symbols. There are two ways to assign a value and type to a symbol. The first is to write the symbol as a label. The label will be assigned the current value and type of the location counter. The second is through the use of the **.set** assembler directive. An arbitrary value and type can be assigned with this directive.

Constants

A constant is an object of ABSOLUTE type and fixed value. The size and appropriate number of digits are controlled by the generation pseudo-ops **.byte**, **.half**, and **.word**. A constant may be one of the following:

- A decimal constant is represented by a contiguous string of the digits 0–9, beginning with a nonzero digit. Examples of decimal constants are:

  123 75 1943 2

- An octal constant is represented by a contiguous string of the digits 0–7 beginning with a zero digit. Examples of octal constants are:

  077 0123 06 037777777777

- A hexadecimal constant is represented by a contiguous string of the digits 0–9 and the letters a-f or A-F, prefixed by Ox or OX. Examples of hexadecimal constants are:

  0x3f 0X9aC 0xabcd 0XFE

Note: Floating point operations and declarations are *not* supported by the processor, but are available in some applications. If supported, floating point constants have the same syntax and interpretation as floating point constants in the C language with the exception that the constant may be preceded by an optional minus (–) sign indicating a negative constant. The precision of the constant (single or double) is always determined by its context.

In order to be recognized as floating point, a constant must contain either a decimal point or one of the exponential characters (e or E). Floating point constants that cannot be encoded exactly in the specified form are rounded off.

Examples of floating point data types are:

31.0500 –16. 0.1024e4 500e–3

Floating point data specifications are expected to conform to the IEEE standard for binary floating-point arithmetic.

Location Counter

The symbol . (read as dot) is the location counter used during the assembly of a program and is reserved for use by the assembler. The type of this symbol is either TEXT, if code is currently being generated for the **.text** section, or DATA, if code is currently being generated for the **.data** section. The initial type of the location counter is TEXT and the initial value is zero.

The location counter represents the address of the next available byte for the placement of assembled code or data, and can change in the following ways:

- as a result of the **.text, .data, .set, .zero, .align, .byte, .half,** or **.word** pseudo-ops
- as a result of the generation of code for a machine instruction.

In the first case, the change is explained in the description associated with each pseudo-op. In the second case, the location counter is incremented by the size of the assembled code *after* the statement is completely assembled.

For each section (**.text, .data,** or **.bss**) there exists a saved location counter value. Initially each saved location counter value is zero. When the programmer issues a section change pseudo-op, the current location counter (i.e., the section being changed from) is saved. The current location counter is then assigned the value of the location counter for the destination section.

Registers

Registers 3 through 8, which are referred to by the assembly language syntax %r3, %r4, %r5, ..., %r8, are the general purpose registers that are always available to the programmer. Registers 0, 1, and 2 are considered general purpose, but have implicit definitions because of certain conventions of the C language. For example, r0 should always be used to return the value of a function. If a floating point double value is returned from a function, it is stored in r0 and r1. If a function returns a structure, then the pointer to that structure should be returned to r2. In general, r0, r1, and r2 are scratch registers. Data transfer instructions **MOVBLW, STRCPY,** and **STREND** also implicitly use these three registers as do the system instructions **MVERNO, INTACK, ENBVJMP, DISVJMP, GATE, RETPS** and **CALLPS.**

Registers 9 (frame pointer), 10 (argument pointer), and 12 (stack pointer) are also implicitly used, in this case by call and return instructions. These registers can be referred to by the assembly language syntax %fp, %ap, and %sp, respectively.

Registers 0, 1, 2, 9, 10, and 12 may be used in any addressing mode, privileged or nonprivileged. The use of r0, r1, and r2 for function calls and returns is described in **5.2.2 Function Calling Sequence.**

The program counter (r15) is a special register that does not work in all addressing modes. The three registers not yet discussed are privileged, and any attempt to write them when the processor is not at kernel execution level results in a privileged register exception. These three registers are the interrupt stack pointer (ISP), the process control block pointer (PCBP), and the process status word (PSW).

The PSW (r11) contains four condition bits — N,Z,V, and C. Because of the pipelining architecture of the processor, the condition codes in the PSW may not be valid immediately after the execution of an instruction. This inherent delay is not a problem for any conditional branch instructions because they wait until the condition codes are valid before testing them. However, if the PSW is read by any non-machine independent instruction,

a NOP instruction should be inserted between the instruction affecting the condition codes and the instruction trying to read the PSW to allow sufficient time for the condition codes to settle.

Note: The assembler supplies the NOP, if needed, for macro ROM instructions.

Executable Instructions

Mnemonics for processor instructions use uppercase letters and machine independent instruction mnemonics use lowercase letters. When coding in assembly language, this distinction must be maintained. Therefore, all machine-specific mnemonics *must* be coded in uppercase, while mnemonics common to the machine independent instructions must be coded in lowercase.

Be careful when switching between processor and machine independent instructions. Although the mnemonics are identical in many cases, the operations are not. For example, the machine independent instruction cmpw &1,&2 will set the less than flag, while the processor instruction CMPW &1,&2 would, under the same conditions, set the greater than flag, because the operand order is reversed.

The processor instruction set is more complete than the machine independent instruction set, but is machine dependent. Machine independent instructions can be portable.

Because floating point operations are not supported by the processor, use of floating point instructions results in a run-time exception. However, these instructions become legal in applications that support floating point operations.

In many cases, the mapping of machine independent instructions to processor instructions is obvious, particularly when synonymous instructions exist in both instruction sets. However, the mappings of machine independent instructions to corresponding processor instructions can be obscure. In particular, there is only a rough correspondence between machine independent instruction set jumps and processor branch instructions. The machine independent instruction set also has four push instructions and several unsigned instructions that have no synonyms in the processor instruction set.

Mappings can be obscure, not only from the lack of equivalent instructions, but also from the considerable changes that are made as part of optimization. About half of the mappings change when optimization is performed. Hence, the only way to determine the mappings is by studying a disassembly.

The MOVEs of the two instruction sets also have a complex mapping. MOVEs can perform conversion from one data type to another. Sign extension, if necessary, is determined by the type of the source. Signs are extended if the source is signed byte or signed halfword. Zero-extension is performed if the source is unsigned byte or unsigned halfword.

The basis for the mappings of MOVE instructions is:

```
movbbh  →  MOVB  {sbyte}src,{shalf}dst
movbbw  →  MOVB  {sbyte}src,{sword}dst
movbhw  →  MOVH  src,{sword}dst
movzbh  →  MOVB  src,{shalf}dst
movzbw  →  MOVB  src,{sword}dst
movzhw  →  MOVH  {uhalf}src,{sword}dst
movthb  →  MOVH  src,{sbyte}dst
movtwb  →  MOVW  src,{sbyte}dst
movtwh  →  MOVW  src,{shalf}dst
```

If the dst operand is a register, the three truncate instructions are:

```
movthb  →  ANDH3  &0xff,src,{byte}dst
movtwb  →  ANDW3  &oxff,src,{byte}dst
movtwh  →  MOVW   src,dst;MOVH dst,dst
```

The notations used in the above mappings are:

```
src - source        u    - unsigned     half - 16-bit data
dst - destination   byte - 8-bit data   word - 32-bit data
s   - signed
```

Operands

The operand and address modes in assembly language are determined by the syntax. The kinds of operands are:

- Basic
- Effective address
- Offset

The basic operand can be used as either a source or destination. The effective address operand is used as a source. The offset is used as a destination. The basic and effective address operands are described by operand descriptors. However, offset is not described by a descriptor. Basic operands read or write a specified location. Effective address operands contain the source address in the instruction. The offset is a signed 8- or 16-bit displacement from the program counter. The resulting address serves as the target for a branch instruction.

Table 5-4. Address Modes

Mode	Syntax	Mode Field	Register Field	Total Bytes	Notes
Absolute					
Absolute	*$expr*	7	15	5	—
Absolute deferred	*$*$expr*	14	15	5	—
Displacement (from a Register)					
Byte displacement	*expr*(%r*n*)	12	0—10,12—15	2	—
Byte displacement deferred	**expr*(%r*n*)	13	0—10,12—15	2	—
Halfword displacement	*expr*(%r*n*)	10	0—10,12—15	3	—
Halfword displacement deferred	**expr*(%r*n*)	11	0—10,12—15	3	—
Word displacement	*expr*(%r*n*)	8	0—10,12—15	5	—
Word displacement deferred	**expr*(%r*n*)	9	0—10,12—15	5	—
AP short offset	*so*(%ap)	7	0—14	1	1
FP short offset	*so*(%fp)	6	0—14	1	1
Immediate					
Byte immediate	*&imm8*	6	15	2	2,3
Halfword immediate	*&imm16*	5	15	3	2,3
Word immediate	*&imm32*	4	15	5	2,3
Positive literal	*&lit*	0—3	0—15	1	2,3
Negative literal	*&lit*	15	0—15	1	2,3,5
Register					
Register	%r*n*	4	0—14	1	1,3
Register deferred	(%r*n*)	5	0—10,12—14	1	1
Special Mode					
Expanded operand type	{*type*}*opnd*	14	0—14	2-6	4

Notes

1. Mode field has special meaning if the register field is 15; see **Absolute** or **Immediate mode**.
2. Mode may not be used for a destination operand.
3. Mode may not be used if the instruction takes effective address of the operand.
4. *type* overrides instruction type; *opnd* is any of the other valid address modes and becomes the real address mode. For total bytes, add 1 to byte count for address mode determined by *opnd*.
5. Negative quantity; overrides expanded operand type and instruction type.

Each operand descriptor identifies the location of the operand. An operand descriptor may be one or more bytes. The format of the first byte of a descriptor is:

mmmmrrrr

where **rrrr** is the register field (bits 0—3) and represents one of r0—r15. The mode field, **mmmm**, is comprised of bits 4—7 and represents the addressing mode. Table 5-4 can be used to determine the proper syntax and mode based on the value of the mode field.

Unless otherwise specified by the instruction, all operands are addressed by a descriptor. The value of the PC is the address of the first byte of the instruction and retains that value for all operand evaluations during the instruction.

Note: Data in the instruction stream may not be ordered the same way that data is ordered when fetched into the processor. In the instruction stream, the byte order is right-to-left; that is, the first byte of the data stream is always the least significant byte. For example, the first byte of a 32-bit immediate value represents bits 0—7 of the operand. The second byte represents bits 8—15; the third byte, bits 16—23; and the fourth byte, bits 24—31.

Expressions

An expression is a sequence of operands separated by operators. An operand is either a constant, a symbol, or an expression enclosed in parentheses.

Expressions can be used as operands to assembler directives and machine instructions, as appropriate. All operators are fundamentally binary in nature. The operator "—" may be used as a unary operator with the interpretation 0—. For example, —x is interpreted as (0—x).

All operators are assumed to be of EQUAL precedence. If anything other than left-to-right evaluation is desired, parentheses must be used for grouping.

If, in the process of evaluating an expression, an intermediate result will not fit in 32 bits, the final value of that expression will be undefined.

The following operators are available:

\+ Produces the 2's complement sum of its operands. One operand *must* be type ABSOLUTE — the other can be any type. The sum has the type of the other operand. All other combinations of operands are illegal.

— Produces the 2's complement result of subtracting the right operand from the left operand. If the right operand is ABSOLUTE, the difference has the type of the left operand. Otherwise, both operands must be of the same type (which cannot be UNDEFINED), and the result has type ABSOLUTE. All other combinations of operands are illegal.

The result of the subtraction can be erroneous when taking the difference between two relocatable symbols. For example, the value of lab1—lab2, where lab1 and lab2 are labels that are both of type TEXT, DATA or BSS, may change due to various

optimizations of the code between lab1 and lab2 that are made after the assignment of values and types to lab1 and lab2. In such cases, the value of lab1−lab2 will not correctly indicate the distance between lab1 and lab2.

* Produces the 2's complement product of its operands. It requires both operands to be of ABSOLUTE type and produces an ABSOLUTE result.

/ Produces the 2's complement quotient of the left operand divided by the right operand. Uneven divisions result in the integer that is the result of truncating the quotient toward zero; for example, 5/−2 = −2. The quotient operator requires both operands to be of ABSOLUTE type and produces an ABSOLUTE result.

Assembler Directives

An assembler directive is a command to the assembler that does not necessarily generate any code. Directives are distinct from executable instructions, that contain mnemonics for machine operations. Every assembler directive is coded as a pseudo-operation (pseudo-op) code followed by zero or more operands. All assembler directives begin with a period (.). Table 5-5 lists all pseudo-ops alphabetically.

Section Control Pseudo Operations. These pseudo-ops provide a method of changing the section in which code is generated and the section in which labels are defined. They work as follows: each of the sections **.text**, **.data**, and **.bss** has its own hidden dot or location counter that indicates where the next code is to be generated for that section. The actual symbol "." starts out with a type of TEXT and a value of zero. Whenever a section control pseudo operation is encountered, the value of dot is stored away into whichever hidden dot is indicated by its type. The value of some other hidden dot is then retrieved and stored as the value of the symbol ".", and the type of dot is set depending on which hidden dot is used.

The following section control pseudo operations are recognized:

.text
.data
.bss *symbol,size,align*

where:

.text causes the current location counter to be saved and then assigned the value of the location counter for the text section. The type of the current location counter is set to TEXT.

.data causes the current location counter to be saved and then assigned the value of the saved value of the location counter for the data section. The type of the current location counter is set to DATA.

.bss causes the bss location counter to be advanced to a multiple of *align* (which must be an ABSOLUTE expression with a value of 2 or 4), and assigns to *symbol* the type BSS and the current value of the bss location counter. The *.bss* section then advances its dot by the value of *size*. *size* refers to the number of bytes; it must be greater than or equal to 0 and have type ABSOLUTE. The type and value of the current location counter remain unchanged.

Table 5-5. Alphabetical List of Pseudo-Operations	
Name	**Operation**
.align *expr*	Increment the current location counter to a multiple of *expr*. *expr* must evaluate to an ABSOLUTE of 2 or 4.
.bss *sym, size, align*	Define the symbol name *sym* in the **.bss** section, and add *size* to the value of dot and **.bss** after aligning it to a multiple of *align*. This does NOT change the current section to **.bss**. *size* must be an ABSOLUTE value and *align* must be an ABSOLUTE value of 2 or 4.
.byte *val*[, *val*]...	Generate initialized bytes containing the 8-bit value *val* in the current section.
.data	Change the current section to **.data**.
.def *name*	Start of the symbolic description for the symbol *name*.
.dim *expr*[, *expr*]...	If the *name* in **.def** is an array, then the expression gives the dimensions. Up to five dimensions are accepted. The type of each expression should be ABSOLUTE.
.endef	Ending bracket for **.def**.
.file "name"	Pass the *UNIX* System source file *name* to the assembler. Only one **.file** is allowed per assembly file.
.global *name*	Treat *name* as a global symbol, equivalent to storage class *extern* in the C language.
.half *val*[, *val*]...	Generate initialized halfwords containing *val* in the current section. Each *val* must be a 16-bit value.
.il	Indicates that a procedure has been expanded in line.
.line *expr*	Define the source line number of the definition of block symbol "name" in **.def**. *expr* should yield an ABSOLUTE value.
.ln *line*[, *addr*]	Create an entry in the line number table for a section. The current dot becomes the default for *addr*. The type of *addr* tells which section owns the line number. The operand *line* should be an ABSOLUTE value of the source line number.
.scl *expr*	Within **.def** give *name* the storage class of *expr*. The type of *expr* should be ABSOLUTE.
.set *name,expr*	Set the value of the symbol *name* to **expr**; *name* must be a symbol.
.size *expr*	If *name* of **.def** is an object such as a structure or an array, assign it size *expr*. The type of *expr* should be ABSOLUTE.

Table 5-5. Alphabetical List of Pseudo-Operations (Continued)	
Name	**Operation**
.tag *str*	If *name* of **.def** is a structure or union, *str* should be the name of that structure or union tag as defined in the previous **.def-.endef** pair. The operand *str* must be a symbol.
.text	Change the current section to **.text**.
.type *expr*	Within a **.def**, give *name* the C compiler type representation *expr*. The type of *expr* should be ABSOLUTE.
.val *expr*	Within **.def**, give *name* the value *expr*. The type of *expr* should be ABSOLUTE.
.word *val*[, *val*]...	Generate initialized words containing *val* in the current section. Each *val* must be a 32-bit value.
.zero *size*	Advance the location counter by *size* and put zeros in the area skipped. The type of *size* should be ABSOLUTE. This pseudo-op is legal only in a **.data** section.

Pseudo Operations Dealing With Symbols. The pseudo-op **.globl** is used to declare that a symbol is to be accessed by more than one program (i.e., given the EXTERNAL attribute). The format is:

.globl *symbol*

This statement has one of two effects:

- If *symbol* is defined in the program in which the **.globl** statement appears, a symbol table entry will appear in the object file that will allow other programs to access *symbol*.
- If *symbol* is not defined in the program in which the **.globl** statement appears, then references to *symbol* will be treated as references to something defined externally. This use of **.globl** is entirely optional since any symbol that is undefined in a program will be assumed to be external.

It is important to note that **.globl** does *not* define the symbol. This pseudo operation is similar to the "extern" declaration in the C language. A symbol is defined either when it is used as a label, when it is used in one of the data generating operations or when it is given a value in an assignment statement. A **.globl** pseudo-op is used on line 9 of the example in **5.2.4 Programming Example**.

Assignment Pseudo Operation. A symbol may be given an arbitrary value and type through the use of the **.set** pseudo-op. It has the form:

.set *symbol, expression*

The *expression* is evaluated and its value and type are assigned to *symbol*. Every symbol that appears in *expression* must either be defined or have the EXTERNAL attribute. Lines 30 and 31 of the example in **5.2.4 Programming Example** show the use of **.set** pseudo-ops.

Assignments are performed during the assembler's first pass over the input program. This has several important consequences:

- The **.set** pseudo-op does not allow forward referencing; i.e., every symbol that appears in *expression* must be defined prior to the assignment statement. Forward references are allowed in other contexts because all other expressions are not evaluated until later passes.
- The result of the assignment may be different from the expected result. For example, consider the assignment

    .set *abc,lab1−lab2*

where *lab1* and *lab2* are labels appearing in the **.text** section. An ABSOLUTE value is assigned to *abc*, which is the distance from *lab2* to *lab1*, during the first pass. This distance may change during subsequent passes if there are offsets between *lab2* and *lab1* that need to be altered. For example, the **jmp** instruction can assemble into a short form (2 bytes) or a long form (3 bytes) depending on the value of the offset. The first pass of the assembler assumes that the 2-byte form can be used. This will be expanded to the 3-byte form if a subsequent pass determines that the label is out of the range for a short jump. This expansion will not be reflected in the value of *abc* if the **jmp** occurs between *lab1* and *lab2*.

Other assignments may have no problem at all. For example, expressions containing only ABSOLUTE operands always yield the correct result. Assignments such as

.set *xyz,lab1*

where *lab1* is a label in the **.text** section, also behave as desired. When code is modified, the assembler changes the values of labels to point to the correct locations. If the value of *lab1* changes, so will the value of *xyz*, because both are TEXT symbols with the same value.

Assignment to Dot. Null data may be generated by assignment to the location counter. The location counter is represented by the dot symbol (.). Assignment to dot may be performed under the following conditions:

- The result type of the expression to be assigned to dot has the same type as dot.
- The value of the expression to be assigned is not less than the value of dot.

If the assignment increases the value of dot by N, then N bytes of null data are generated. Assignment to dot is most often used to provide holes or spaces in code. For example, the statement

.set .,.+10

generates 10 bytes of null data. The assembler defines null data in the **.text** section as NOPs (0x70); null data in the data section is zero.

Alignment Pseudo Operation. The alignment pseudo-op **.align** causes the next data item or instruction to be assembled at an address that is a multiple of 2 or 4. It has the form

.align *expression*

where *expression* must evaluate to an ABSOLUTE 2 or 4. A **.align 2** causes the value of current location counter to be incremented by one if its current value is not a multiple of 2. A **.align 4** causes the value of the current location counter to be incremented by one, two or three, if its current value is not a multiple of four. The appropriate increment (one, two, or three) needed to bring the location counter to a multiple of four is chosen. If this directive is used in the **.text** section, any space skipped will be filled with NOP instructions. If it is used in the **.data** section, any space skipped will be filled with zeros.

Data Generation Pseudo Operations. Data generation pseudo-ops are used for declaring variables. The data generation pseudo operations — **.byte**,**.half**, and **.word** generate 8-, 16-, and 32-bit integer constants, respectively. The forms are

.byte *expr*, ...
.half *expr*, ...
.word *expr*, ...

Each expression will be converted into its perspective data type. The location counter must be properly aligned with **.align** before each use of one of these pseudo-ops. Dot is then incremented by one, two, or four (depending on the pseudo-op) after the generation of each data item in the list of expressions for that statement. For example, **.word** .,.,. generates three words of data and each word contains the address of the first byte of that word. Therefore, each word contains a different value.

Each expression may be given a bit width by prefacing it with an integer constant followed by a colon. This format for bit width is

n:*expr*

where n ranges from 0 to 8 for **.byte**, 0 to 16 for **.half**, and 0 to 32 for **.word**. Nonprefaced expressions have an assumed bit width of 8, 16, or 32, depending on whether the **.byte**, **.half**, or **.word** pseudo-op is used. The expression, which must be ABSOLUTE, is converted into the proper representation and placed in a field of the indicated width.

For example,

mode: **.byte** 5:x+y, 3:0

initializes an 8-bit variable, *mode*, by setting the upper five bits of *mode* to the result of the expression x + y, and the lower three bits to zero.

Fields are assigned from high order bit positions (i.e., bit 7 of a byte) to low-order bit positions. Each successive expression is placed into a field that begins with the next lower bit position. The location counter is adjusted after the generation of each data item; it always indicates the address of the first *byte* into which the current data item is to be placed.

A field is not allowed to cross the implied boundary indicated by one of the above pseudo-ops. If too few fields are encountered to fill the indicated unit of memory, enough zeros are supplied to fill the low order bits.

The data generation pseudo-op **.zero** allocates an area of memory and fills it with zeros. It has the form

.zero *size*

where *size* is the number of bytes to allocate and fill with zeros. The **.zero** pseudo-op advances the location counter by *size* and puts zeros in each byte of memory that is skipped. It is legal only in the **.data** section. Variables declared static in a C source program are assembled through this pseudo-op.

Symbolic Debugging Pseudo Operations. Symbolic debugging pseudo-ops are provided for making entries in the symbol and line number tables in the object file. The presence of symbolic debugging pseudo operations in an assembly language program has no effect on program execution. These statements merely serve to transparently pass information from the user code to the symbolic debugger.

The basic symbolic debugging pseudo operations are **.def** and **.endef**. These are used as a pair to surround a list of pseudo operations that assign attributes to a symbol. The format used is:

.def *name*
.
.
{Attribute-assigning pseudo operations}
.
.
.endef

The attribute-assigning pseudo operations between **.def** and **.endef** assign attributes to the symbol *name*. These attribute-assigning pseudo operations are available:

.val *expr*	Gives the value *expr* to the symbol *name*. In general, the type of *expr* (TEXT, DATA, etc.) is used to determine the section with which the symbol *name* is associated.
.scl *expr*	Declares a storage class for the symbol *name*. *expr* must yield a value of ABSOLUTE type that corresponds to the portable C compiler's internal representation of a storage class.
.type *expr*	Declares a data type for the symbol *name*. *expr* must yield a value of ABSOLUTE type that corresponds to the portable C compiler's internal representation of a type and derived type.
.tag *str*	Used when *name* is a C level structure or a union. *str* is a structure or union tag that is defined by some other **.def-.endef** pair.
.line *expr*	Used when *name* is a block symbol. *expr* yields a value of ABSOLUTE type that gives the line number of the declaration for *name*.

.size *expr*	Used when *name* is a C level structure or an array that does not have a predetermined size. *expr* should yield a value of ABSOLUTE type that gives the size of *name*, usually in bytes, or in bits if *name* is a bit field.
.dim *expr1,expr2,...*	Used when *name* is an array. Each expression yields a value of ABSOLUTE type that gives the corresponding dimension of the array. Since the *UNIX* System implementation of the C language supports up to five dimensions for an array, there may be up to five arguments to the **.dim** pseudo-op.
.il	Used to indicate that a procedure has been expanded in-line.

For symbolic debugging purposes, the order of symbols is very important. The assembler has no knowledge of this ordering; it just passes the symbols through from the C compiler so they may be accessed by the symbolic debugger.

As with **.globl**, the **.def** pseudo-op does not define the symbol. A symbol table entry is created but no definition occurs.

File Name Pseudo Operation. Associated with each assembly file can be at most one **.file** pseudo-op. It has the form

.file "*name*"

where "*name*" is a double-quoted string of 1 to 14 characters. This pseudo-op is normally used to pass the name of the C source file from which the assembly program originated. *name* then becomes part of the symbol table and can be accessed at run time. Line 1 of the example in **5.2.4 Programming Example** demonstrates the **.file** pseudo-op.

Line Number Pseudo Operation. Each section in the object file has a line-number table associated with it that maps line numbers in the source code to addresses within the section. A line-number entry may be made using the **.ln** pseudo operation as:

.ln *line*[,*value*]

The operand *line* must have a value of ABSOLUTE type that gives a line number in the source code. The optional operand *value*, if present, must have a value of type TEXT, DATA, or BSS that gives the address within the section where the line number occurs. If the *value* operand is missing, the value of the current location counter will be used as the address of the line number.

Function Calling Sequence

The *WE* 32100 Microprocessor C language stack frame and calling sequence are discussed in this section. This information is intended for those who require a detailed knowledge of the implementation of C function calls or need to perform assembly language function calls. The stack frame is examined, paying particular attention to the size and location of its contents. An example of a typical function call is given, describing the needs of the called and calling programs. High-level code that depends on these implementation details should be avoided.

Four registers are manipulated as part of each function call. These are the frame pointer (FP), r9; the argument pointer (AP), r10; the stack pointer (SP), r12; and the program counter (PC), r15.

The frame pointer and argument pointer are only affected by the function call and return instructions. In C language, the frame pointer points to the location in the stack that is the start of the area containing local variables for that function. The argument pointer points to the location in the stack that contains the first of the set of arguments for that function.

The calling sequence is presented as if the C compiler were implementing the function call, i.e., assembler instructions have been generated, and **m32as** is translating to the processor instruction set. Two machine independent instructions, call and save, establish the calling environment and one machine independent instruction, ret, unwinds it. If the corresponding processor instructions were being used, the CALL and SAVE instructions would establish the calling environment, and two instructions RESTORE and RET would be required to conclude the function properly. Thus the processor instruction set requires two instructions as opposed to one assembler instruction for this task. The machine independent instructions give an additional degree of control in the calling sequence, while the processor instructions have a closer interaction with the CPU. The processor implementation of these instructions is described in the **Stack Frame** section below. The four affected registers are initially set as:

1. **PC.** The program counter is set to the address of the first executable instruction of the calling program.
2. **SP.** The stack pointer points to the top of the stack and is properly set so that a new procedure may be called.
3. **FP.** The frame pointer points just past the top of the register save area. The register save area is a FIXED size region created on the stack by the function call instructions for saving registers. Just past the register save area is a stack region reserved to store temporary (also called automatic) variables for a function.
4. **AP.** The argument pointer points to the BEGINNING of a list of arguments to the function.

The stack frame reserves space for six registers in addition to the PC, FP, and AP. These six registers correspond to the six registers available as register variables in C programs.

Note that the PC, FP, and AP are always pushed on the stack in a function calling sequence; the SP is not because its value is always implicit.

Although space is reserved on the stack for up to six registers plus the AP and FP, only the AP and FP MUST be pushed. The remaining six user registers should be pushed (via the save instruction) only when necessary.

Stack Frame. A stack frame is created at run time for each instance of a function call. The frame is destroyed when the called function returns to the calling function. Each stack frame contains the information needed to restore the calling function to its prior state (i.e., the state it was in before it made the function call). The stack frame also contains the arguments passed to it by the caller, space for its automatic variables, and space for any temporary variables needed during execution. The stack begins at lower addresses and

grows upward to higher addresses. Figure 5-3 shows a diagram of a typical stack frame.

Actions of Calling Function. To make a function call, the calling function must first push all of the called functions onto the stack. The arguments are pushed in the same order as they appeared in the function call. Every argument must be pushed on the stack as a 32-bit quantity. Characters must be converted to integers, and structures of uneven length must be filled out to word boundaries, even though the last byte(s) are meaningless. Also, multiple word arguments, such as structures, will require multiple pushes.

For example, the following section of code implements a C level function of the form

func *(A,B,C)*

where *A* and *C* are integer arguments and *B* is a character (byte) argument. The machine independent code to call func is:

```
.
.
.
pushw     A          #extend byte to 32 bits
pushzb    B
pushw     C
call      &3,func    #call the function, specifying
                     #the number of arguments
.
.
.
```

The equivalent processor instructions are:

```
PUSHW     A
PUSHW     {ubyte}B
PUSHW     C
CALL      -12(%sp),func
```

The last statement is a call to the desired function, thus transferring control to the called function. This process is accomplished by the machine independent call instruction. Figure 5-4 shows the stack after the call has been executed.

Actions of Called Function. The called function completes the initialization started by the caller. The first responsibility of the called function is to use the save instruction to implement a C procedure frame. The save instruction can save up to six registers (r3 through r8) so they may be used by the function. After saving the specified number of registers, the save instruction adjusts the stack pointer and frame pointer to point beyond the end of the fixed-size register save area. After executing a save instruction specifying that five registers should be saved, the stack would look like Figure 5-5.

The remaining responsibility of the called function is to allocate space for the automatic and temporary variables it will use. The function does this by adding a constant to the stack pointer. This leaves %sp pointing somewhere above %fp in the stack. The stack frame then appears as shown on Figure 5-3. Only after this has been accomplished should

the called function begin to execute.

In the above example of the call to function func, the called function should have the machine independent syntax:

```
        .
func:   save    &5          #save the caller's registers
        addw2   &8,%sp      #allocate stack space for the
          .                   automatic and temporary variables
          .
          .
{function body}
          .
          .
        ret     &5          #restore registers and return to caller
```

The equivalent processor instructions are:

```
func:   SAVE    %r4
        ADDW    &8,%sp
          .
          .
        RESTORE %r4
        RET
```

To return to the caller, a function should execute an machine independent return (RET) instruction. This is mapped into processor RESTORE and RET instructions. The RESTORE instruction is the inverse of the SAVE instruction; i.e., it restores up to six registers and the frame pointer. After the RESTORE instruction, the stack is as described after the call instruction (see Figure 5-4).

Note: The number of registers to be restored MUST be the same as the number of registers saved; otherwise the results are undefined.

The RET instruction is the inverse of the call instruction; i.e., RET returns the stack to the state it was in before the function call.

When the routine accesses local data in the stack, it must do so by offsets from the frame pointer. A routine accessing data passed to it as an argument must use offsets from the argument pointer.

Locations in the stack area above the stack pointer are not protected from being destroyed by interrupting processes and should not, therefore, be used without first incrementing the stack pointer. The push instruction provides a convenient way of allocating stack space a word at a time. In cases where speed is critical and a large number of words are to be stored, it may be more efficient to allocate the total size of the area needed with a single add to the stack pointer.

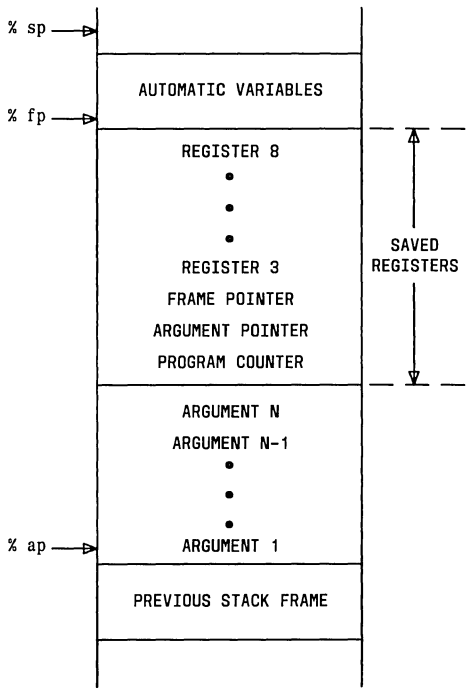


Figure 5-3. Typical Stack Frame for a Function Call

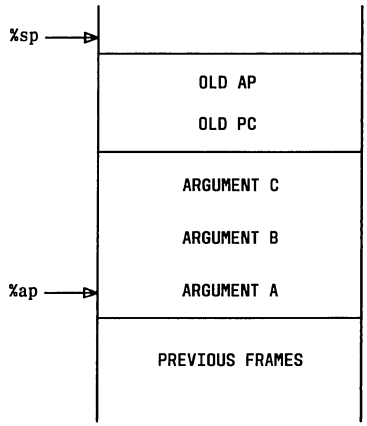


Figure 5-4. Stack Frame Following a Call Instruction

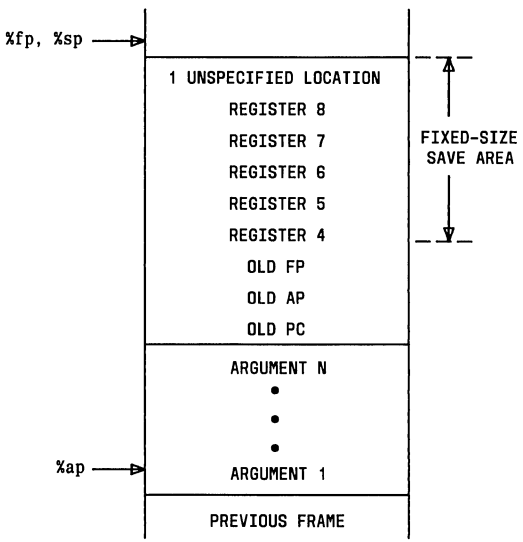


Figure 5-5. Stack Frame After Three Registers are Saved

The SP should never be modified directly except with the push and pop instructions. These two instructions automatically increment or decrement the stack pointer. If a program took such action directly, and care was not taken, the contents of the restored registers could be destroyed by subsequent stack manipulations.

5.2.3 Exception Conditions

Several kinds of events may occur that will interrupt the execution of a program. These may either be internally generated, that is, recognized and generated by the processor, or externally generated, such as an I/O interrupt for a memory fault.

5.2.4 Programming Example

Following is an example of the compiler output and the assembler output for the function *prefix*. The *prefix* function consists of C language code that determines if one string is a prefix of another.

The example includes many of the pseudo-ops explained in this chapter. These pseudo-ops form the prologue and epilogue sections that the compiler always generates. The M4 processor can provide these sections for assembly language programs if the **−m** option is specified and the defined macros are used.

This example shows a program that was compiled, but not optimized. Therefore, the assembly code contains #REGAL statements that were inserted by the compiler for use by the optimizer, but never used. Since these lines have the format of assembler comments, they are simply ignored by the assembler.

Line numbers have been added for convenience; otherwise, the left column presents all of the machine code produced by the **m32cc** compiler. The right column presents the corresponding C language statements. The correspondence between C code and assembly code can be seen for if and while statements.

```
                Assembly Code                    C Language Statement
 1.     .file    "prefix.c"          prefix(a,b)
 2.     .data
 3.     .text
 4.     .align   4
 5.     .def     prefix;
        .val     prefix;
        .scl     2;                  char *a,*b;
        .type    044;                {char *p;
        .endef
 6.     .globl   prefix
prefix:
 7.     save     &.R1
 8.     addw2    &.F1,%sp
 9.     movw     4(%ap),4(%fp)       char *q=b;
10.     movw     0(%ap),0(%fp)       p=a;
11.     jmp      .L30
```

```
.L31:
12.         addw2   &1,0(%fp)           p++;
13.         addw2   &1,4(%fp)           q++;
14.         cmpb    *0(%fp),*4(%fp)     if(*p!=*q)
15.         je      .L32
16.         movw    &0,%r0                  return(0);}
17.         jmp     .L28
.L32:
.L30:
18.         cmpb    *0(%fp),&0          while((*p!=NULL)
19.         je      .L33                &&
20.         cmpb    *4(%fp),&0              (*q!=NULL)) {
21.         jne     .L31
.L33:
.L29:
22.         movw    &1,%r0              return(1);
23.         jmp     .L28
#REGAL      0       NODBL
#REGAL      48      AUTO                0(%fp)
#REGAL      48      AUTO                4(%fp)
.L28:
24.         .def    .ef;
            .val    .;
            .scl    101;
            .line   10;
            .endif
25.         .ln     10
26.         .set    .F1,8
27.         .set    .R1,0
28.         ret     &.R1
29.         .def    prefix;
            .val    .;
            .scl    -1;
            .endif
30.         .data
```

A disassembly of the assembler output below shows the processor instructions for this routine. Note that the function saves no registers, and therefore starts with SAVE %fp. The assembler directives have been omitted.

```
section     .text
prefix()
            SAVE        %fp
            ADDW2       &0x8,%sp
            MOVW        0x4(%ap).0x4(%fp)
```

```
MOVW    0(%ap),0(%fp)
BRB     0x11 <20>
INCW    0(%fp)
INCW    0x4(%fp)
CMPB    *0x4(%fp),*0x0(%fp)
BEB     0x6 <20>
CLRW    %r0
BRB     0x11 <2f>
TSTB    *0x0(%fp)
BEB     0x7 <2a>
TSTB    *0x4(%fp)
BNEB    -0x17 <11>
MOVW    &0x1,%r0
BRB     0x2 <2f>
RESTORE %fp
RET
NOP
NOP
```

This listing was actually produced by disassembling the object file *prefix.o* with the **m32dis** utility described in **5.5 UTILITIES AND LIBRARY ROUTINES.**

5.2.5 Machine Independent Instruction Set

The machine independent instructions are listed alphabetically by mnemonic in Table 5-6. Many instructions have three forms (byte, halfword and word) that are characterized by a *b*, *h*, or *w* in their names. The term "complex" appearing under the mapping heading indicates that an instruction has a complex (one-to-many) mapping into a sequence of *WE* 32100 Microprocessor instructions. Instructions with simple (one-to-one) mapping map to a corresponding processor instruction with the possibility of an optimized form. If an instruction has an optimized form, the **m32as** assembler will map that instruction into a different hexadecimal encoding than is used for the unoptimized form.

Table 5-6. Machine Independent Instruction Set

Mnemonic	Name	Mapping
acjl	Add, compare, and jump less	Complex
acjle	Add, compare, and jump less or equal	Complex
acjleu	Add, compare, and jump less or equal unsigned	Complex
acjlu	Add, compare, and jump less unsigned	Complex
addb2,addh2,addw2	Add (2 operand) - byte, halfword, word	Simple
addb3,addh3,addw3	Add (3 operand) - byte, halfword, word	Simple
alsw2	Arithmetic left shift (2 operand)	Simple
alsw3	Arithmetic left shift (3 operand)	Simple
andb2,andh2,andw2	AND (2 operand) - byte, halfword, word	Simple
andb3,andh3,andw3	AND (3 operand) - byte, halfword, word	Simple
arsw2	Arithmetic right shift (2 operand)	Simple
arsw3	Arithmetic right shift (3 operand)	Simple
atjnzb,atjnzh, atjnzw	Add, test, and jump not zero - byte, halfword, word	Complex
bitb,bith,bitw	Bit test - byte, halfword, word	Simple
call	Call	Simple
cmpb,cmph,cmpw	Compare - byte, halfword, word	Simple
divw2	Divide (2 operand)	Simple
divw3	Divide (3 operand)	Simple
extzv	Extract field	Simple
insv	Insert field	Simple
jbc	Jump on bit clear	Complex
jbs	Jump on bit set	Complex
je	Jump equal	Simple
jg	Jump greater	Simple
jge	Jump greater or equal	Simple
jgeu	Jump greater or equal unsigned	Simple
jgu	Jump greater unsigned	Simple
jl	Jump less	Simple
jle	Jump less or equal	Simple
jleu	Jump less or equal unsigned	Simple
jlu	Jump less unsigned	Simple
jmp	Jump	Simple
jne	Jump not equal	Simple
jneg	Jump negative	Simple
jnneg	Jump not negative	Simple
jnpos	Jump not positive	Simple

Table 5-6. Machine Independent Instruction Set (Continued)		
Mnemonic	**Name**	**Mapping**
jnz	Jump not zero	Simple
jpos	Jump positive	Simple
jsb	Jump to subroutine	Simple
jz	Jump zero	Complex
llsw2	Logical left shift (2 operand)	Simple
llsw3	Logical left shift (3 operand)	Simple
lrsw2	Logical right shift (2 operand)	Simple
lrsw3	Logical right shift (3 operand)	Simple
mcomb,mcomh,mcomw	Move complemented - byte, halfword, word	Simple
mnegh	Move negated - halfword	Simple
mnegw	Move negated - word	Simple
modw2	modulo (2 operand)	Simple
modw3	modulo (3 operand)	Simple
movaw	Move address	Simple
movb,movh,movw	Move - byte, halfword, word	Simple
movbbh	Move bit extended - byte to halfword	Simple
movbbw	Move bit extended - byte to word	Simple
movbhw	Move bit extended - halfword to word	Simple
movblb	Move block - byte	Complex
movblh	Move block - halfword	Complex
movblw	Move block - word	Simple
movthb	Move truncated - halfword to byte	Simple
movtwb	Move truncated - word to byte	Simple
movtwh	Move truncated - word to halfword	Complex
movzbh	Move zero extended - byte to halfword	Simple
movzbw	Move zero extended - byte to word	Simple
movzhw	Move zero extended - halfword to word	Simple
mulw2	Multiply (2 operand)	Simple
mulw3	Multiply (3 operand)	Simple
orb2,orh2,orw2	OR (2 operand) - byte, halfword, word	Simple
orb3,orh3,orw3	OR (3 operand) - byte, halfword, word	Simple
pushaw	Push address - word	Simple
pushbb	Push bit extended - byte	Complex
pushbh	Push bit extended - halfword	Complex
pushw	Push word	Simple
pushzb	Push zero extended - byte	Complex
pushzh	Push zero extended - halfword	Complex
ret	Return	Complex
rsb	Return from subroutine	Simple
save	Save	Simple

Table 5-6. Machine Independent Instruction Set (Continued)		
Mnemonic	**Name**	**Mapping**
subb2,subh2,subw2	Subtract (2 operand) - byte, halfword, word	Simple
subb3,subh3,subw3	Subtract (3 operand) - byte, halfword, word	Simple
udivw2	Unsigned divide (2 operand)	Simple
udivw3	Unsigned divide (3 operand)	Simple
umodw2	Unsigned modulo (2 operand)	Simple
umodw3	Unsigned modulo (3 operand)	Simple
umulw2	Unsigned multiply (2 operand)	Simple
umulw3	Unsigned multiply (3 operand)	Simple
xorb2,xorh2,xorw2	XOR (2 operand) - byte, halfword, word	Simple
xorb3,xorh3,xorw3	XOR (3 operand) - byte, halfword, word	Simple

5.3 LINK EDITOR

The link editor creates load files by combining object files, performing relocations, resolving external references, and supporting symbolic debugging information. The inputs to **m32ld** are object files produced by either the **m32cc** compiler, the **m32as** assembler, or by a previous **m32ld** run. The link editor combines these input object files to form either a relocatable or an absolute (i.e., executable) object file. The object file format is given in **5.4 OBJECT FILE FORMAT**.

The link editor control language can:

- Specify memory configurations for the intended target system.
- Combine object file segments in several ways and cause them to be loaded at specific addresses or within specific portions of memory.
- Define or redefine global symbols at load time.

5.3.1 Link Editor Command

The link editor is called by the command line

m32ld [options] filenam1 filenam2...

Input files to the link editor must be object files, archive libraries containing object files, or ASCII source files containing link editor directives. An archive library is merely a group of object files that are collected in one place because they are expected to be useful in several applications. The so-called "magic number" (in the first two bytes of the file header) indicates which type of input file has been encountered. If the link editor does not recognize the magic number, it will assume the file is a text file containing **m32ld** directives and will attempt to parse it. Input object files and archive libraries of object files are linked together to form an output object file that is executable on the target system, provided there are no unresolved references. Input source files containing **m32ld** directives are also called *ifiles*. Object files usually have the form name .o, although the link editor does not enforce this convention.

To link object files named file1.o and file2.o, the following command line is sufficient:

m32ld file.o file2.o

No directives or options are needed. If no errors occur, an executable object file named **m32a.out** is created.

The sections of the input files are combined in order. That is, if file1.o and file2.o each contain the standard **.text**, **.data**, and **.bss** sections, then the output file will also contain these three sections. **m32ld** will concatenate the **.text** sections from file1.o and file2.o to form the output **.text** section. The output **.data** and **.bss** sections will be similarly formed.

Instead of entering the names of the files to be link edited and the **m32ld** options on the command line, this information can be placed in an ifile which can be passed to the link editor. For example, if the files file1.o, file2.o, and file3.o were to be frequently linked using the options **−m** and **−V**, the command line would be:

m32ld *−m −V* file1.o file2.o file3.o

Rather than entering this command each time, an ifile can be created containing the statements:

```
−m
−V
file1.o
file2.o
file3.o
```

The link editor can then be invoked using

m32ld *ifilename*

where *ifilename* is the name of the ifile. Some of the object files to be link edited can be specified in an ifile and others on the command line. The same holds true for options — some can come from the command line while others come from the ifile.

Input files are link edited in the order they are encountered, whether they are encountered on a command line or in an ifile. For example, the command line

m32ld file1.o ifile file2.o

can be used with an ifile containing

```
file3.o
file4.o
```

to form an object file with the form:

file1.o file3.o file4.o file2.o

This example demonstrates an important property of ifiles; i.e., they are read and processed as soon as they are encountered in a command line.

Command Line Options

Options may be interspersed with file names both on the command line and in an ifile. Except for the **−l** and **−L** options, all options may be specified in any order. The **−l** option names an input archive library. Like other input files, libraries are searched and link-edited just as they are encountered, so ordering is important. The **−L** option names directories to be searched when looking for an archive library. Therefore, to be effective, a **−L** option must appear before any **−l** options.

A minus sign (−) precedes all options, whether options are specified in an ifile or on the command line. White space, (blanks or tabs) separates arguments to the option from the option letter (except for **−l** and **−L**). Options recognized by **m32ld** are listed in Table 5-7.

Table 5-7. m32ld Command Line Options

Option	Argument	Description
−a	None	Produce an absolute executable file, and give warnings for undefined references. Relocation information is stripped from the output object file unless the **−r** option is invoked. The **−r** option is needed only when an absolute object file must retain its relocation information (an unusual case). The **−a** option is invoked by default, but must be explicitly entered when the **−r** option is in effect.
−e	*epsym*	Set the default entry point address for the output file to *epsym*. This option both defines the entry point symbol and forces the printing of a standard *UNIX* System **a.out** header.
−f	*fill*	Initialize holes within output sections using the argument *fill*. The argument must be a two-byte constant, e.g., **−f**0xdfff.
−l	*lnam*	Link edit the library specified by *lnam*. The library name is interpreted to be **lib***lnam***.a**, where *lnam* can contain up to seven characters. A library is searched when its name is encountered, so the placement of a **−l** option is significant. Location of the library is a SGP build parameter. See Note.
−M	None	Print a warning message for all external variables that are multiply defined.
−m	None	Produce a map or listing of the input/output sections on the standard output.
−o	*outfile*	Name the output object file *outfile*. The name of the default object file is **m32a.out**. See Note.

Note: Argument is appended to option with no embedded blanks.

Table 5-7. m32ld Command Line Options (Continued)		
Option	**Argument**	**Description**
−r	None	Retain the relocation entries in the output object file. Relocation entries must be saved if the output file is to become an input file to a subsequent **m32ld** run. The link editor complains about unresolved references if the **−r** option was omitted. The **−r** option is useful for forming subsystems.
−s	None	Strip line number entries and symbol table information from the output object file. All symbols are removed, including global and undefined symbols. Relocation entries are meaningless without the symbol table, so the **−r** option cannot be used with **−s**.
−t	None	Turn off the check ensuring that all instances of a multiply defined symbol be the same size.
−u	*symname*	Take the argument *symname* as a symbol and enter it as undefined in the symbol table. This is useful for loading entirely from a library, because initially the symbol table is empty and an unresolved reference is needed to force the loading of the first routine. See Note.
−L	*dir*	Search for libraries in the directory *dir* before looking in the default location (LIBDIR). See Note.
−N	None	Put the data section immediately after the text section in the output file.
−V	None	Print (on standard error) a message giving information about the version of **m32ld** being used.
−VS	*num*	Give the version stamp *num* to the **m32a.out** file that is produced. The *num* argument is taken as a decimal number and stored in the standard **a.out** header of the output object file. See Note.
−x	None	Do not preserve local symbols in the output symbol table; enter external and static symbols only. This saves some space in the output file.

Note: Argument is appended to option with no embedded blanks.

5.3.2 Link Editor Command Language

The **m32ld** command language enables the user to control the design of the object module created from the input object files. The language consists of input statements and specifications that can specify memory configuration, bind sections to named addresses or portions of memory, and define global symbols. Additional caution is required when using the power and flexibility of input directives. Any pointer that has the value zero must not point to an object. (The C language defines a null pointer as zero, 0.) To ensure this property, users must not place any object at virtual address zero in the data space.

Expressions

Expressions may contain global symbols, constants, and most of the basic C language operators. As in the C language, constants with a number are recognized as decimal unless preceded with 0 for octal or 0x for hexadecimal. All numbers are treated as long integers. Symbol names may contain upper- or lower-case letters, digits, and the underscore (_). Symbols within an expression have the value of the *address* of the symbol only. **m32ld** will not perform symbol table lookup to find the contents (value) of a symbol, the dimensionality of an array, structure elements declared in a C program, etc.

The link editor uses an input scanner (lexical analyzer) to identify symbols, numbers, operators, etc. The current scanner design makes the following names *reserved* and unavailable as symbol names or section names:

ALIGN	l	PHY
align	len	phy
ASSIGN	length	RANGE
assign	LENGTH	range
BLOCK	MEMORY	REGIONS
block	NOLOAD	SECTIONS
COPY	O	SPARE
DSECT	org	spare
GROUP	origin	TV
group	ORIGIN	

The operators that are supported, in order of precedence from high to low, are:

! ~ − (Unary)
* / %
+ − (Binary)
>> <<
== != > < <= >=
&
|
&&
||
= += −= *= /=

The above operators have the same meaning as in C language. The associativity is also the same as in C language, with left to right associativity except for !, −(unary), =, +=, −=, *=, and /=. Operators on the same line have the same precedence.

Assignment Statements

External symbols may be defined and assigned addresses using the assignment statement. The assignment statement syntax is:

```
symbol = expression;
          or
symbol op = expression;
```

where *op* is one of the operators +, −, *, or /. Assignment statements must be terminated by a semicolon(;).

All assignment statements, with the exception of the one case described in the following paragraph, are evaluated after allocation has been performed and all input-file-defined symbols have been appropriately relocated, but before the actual relocation of the text and data itself. Therefore, if an assignment statement expression contains any symbol name, the address used for that symbol in the evaluation of the expression reflects the symbol address in the *output* object file. However, within text and data, references to symbols given a value through an assignment statement will access this latest assigned value. Assignment statements are processed in the same order as input to **m32ld**.

Assignment statements are normally placed outside the scope of section-definition directives (see **Creating and Defining Symbols at Link-Edit Time** in this section). However, there exists a special symbol, called *dot* or ".", which can occur only *within* a section-definition directive. This symbol refers to the current virtual address of the link editor location counter. Thus, assignment expressions involving "." are evaluated during the allocation phase of **m32ld**. Assigning a value to the "." symbol within a section-definition directive increments and resets the **m32ld** location counter, and can create holes within the section. Assigning the value of the "." symbol to a conventional symbol permits the final, allocated address of a particular point within the link edit run to be saved.

Align is provided as a shorthand notation to allow alignment of a symbol to an n-byte boundary within an output section, where *n* is a power of two. For example, the expression

```
align(n)
```

is equivalent to

```
(.+n−) & ∼ (n−1)
```

Memory Configurations

By default, the link editor considers the target processor to have an address range of 56 kbytes, numbered from 0x100000 to 0x10CFFF. This comprises the virtual address space into which all input files are linked.

To help allocate space, virtual memory is partitioned into *configured* and *unconfigured* memory. By default, all virtual memory is treated as configured, and unconfigured memory is treated as reserved and unusable by the link editor.

Note: Nothing can ever be linked into unconfigured memory. Thus, making a certain memory range unconfigured is one way of marking the addresses in that range as illegal or nonexistent with respect to the linking process.

Memory configurations other than the default must be explicitly set up using the link editor command language.

MEMORY directives are used to specify:

- The total size of the virtual space of the target *WE* 32100 Microprocessor.
- The configured and unconfigured areas of the virtual space.

If no directives are supplied, the link editor assumes that virtual memory is configured for 56 kbytes, beginning at address 0x100000 and ending at address 0x10CFFF.

MEMORY directives can assign an arbitrary name of up to eight characters to a virtual address range. Output sections can then be forced to be bound at virtual addresses within specific *named* memory areas. Memory names may contain upper- or lower-case letters, digits, and the special characters '$', '.', and '_'. Names of memory ranges are only used by the link editor and are not carried in the output file symbol table or headers.

When MEMORY directives are used, all virtual memory not described in a memory directive is considered to be *unconfigured*. Unconfigured memory is not used in the allocation process of **m32ld**, and hence nothing can be link edited, bound, or assigned to any address within unconfigured memory.

As an option to a MEMORY directive, *attributes* may be associated with a named memory area. The attributes permit an output section to restrict where it will be bound; such a section will be bound only to a memory area with the named attributes. The attributes assigned to output sections in this manner are recorded in appropriate section headers in the output file, thus making possible error checking in the future. (For example, putting a text section into writable memory is a potential error condition.) Currently, error checking of this type is not implemented.

Attributes that are currently accepted are:

- R Readable memory
- W Writable memory
- X Executable memory, i.e., instructions may reside in this memory
- I Memory that can be initialized; stack areas are typically not initialized.

Other attributes may be added in the future if necessary. If no attributes are specified on a MEMORY directive, or if no MEMORY directives are supplied, memory areas will assume all the attributes R, W, X, and I.

The syntax of the MEMORY directive is:

```
MEMORY  {
            name1(attr): origin = n1,length = n2
            name2(attr): origin = n3, length = n4
            etc.
        }
```

The keyword *origin* (or *org* or *o*) must precede the origin of a memory range, and the keyword *length* (or *len* or *l*) must precede the length, as shown in the above prototype. The origin operand refers to the virtual address of the memory range. Origin and length are entered as 32-bit constants in either decimal, octal, or hexadecimal (using standard C syntax). Origin and length specifications, as well as individual MEMORY directives, may be separated by either white space or commas.

By specifying MEMORY directives, the user can tell the link editor that memory is configured in some manner other than the default. For example, if it is necessary to prevent anything from being linked to the first 0x10000 words of memory, a MEMORY directive can accomplish this:

```
MEMORY  {
            valid: org = 0x10000, len = 0xfe0000
        }
```

Section Definition Directives

A section is the smallest relocatable unit of an object file and must reside in a contiguous block of memory. Each section has a starting virtual address and a size. Section headers (which are described in **5.4 OBJECT FILE FORMAT**) start each file and contain information describing all included sections. Sections from input files are combined to form output sections containing executable text, data, or a mixture of both. Although there may be "holes" or gaps between input sections and between output sections, contiguous storage is allocated *within* each output section.

SECTIONS directives describe how input sections are to be combined, direct where to place output sections (both in relation to each other, and to the entire virtual memory space), and permit the renaming of output sections.

In the default case where no SECTIONS directives are given, each input section appears in an output section of the same name. For example, if a number of object files from the compiler are linked, each containing the three sections, **.text, .data**, and **.bss**, the output object file will also contain three sections, **.text, .data**, and **.bss**. If two objects files are linked, one containing sections s1 and s2, and the other containing sections s3 and s4, then the output object file will contain the four sections s1, s2, s3, and s4. The *order* of these sections depends on the order in which the link editor saw the input files.

The basic syntax of the SECTIONS directive is:

```
SECTIONS  {
              secname1:  {  file_specifications,
                            assignment_statements
                            }
              secname2:  {  file_specifications,
                            assignment_statements
                            }
              etc.
          }
```

The various types of section definition directives are discussed in the remainder of this section.

Virtual Address and Bindings. All addresses manipulated by **m32ld** are 32-bit absolute addresses defined relative to address zero. The address of a section means the virtual address of the start of the section. The address of a symbol is the virtual address of the text or data word defining the symbol. Physical addresses to the link-editing are equivalent to virtual addresses, i.e., no distinction is made by **m32ld**.

It is often necessary to have a section begin at a specific, predefined address. The process of specifying this starting address is called binding, and the named section is said to be "bound at" or "bound to" the required address. Binding generally refers to output sections, but it is also possible to bind global symbols using an assignment statement from the link editor command language.

File Specifications. Within a section definition, the files and sections of files to be included in the output section are listed as they appear in the output section. Sections from an input file are specified by

```
filename (secname)     or     filename (secnam1 secnam2 ... )
```

Input file sections are separated either by white space or commas, as are the file specifications.

If a filename appears with no sections listed, *all* sections from the file are linked into the current output section. For example,

```
SECTIONS  {
              outsec1:  {
                            file1.o (sec1)
                            file2.o
                            file3.o (sec1,sec2)
                        }
          }
```

links all sections from file2.o into the output.

The order in which the input sections appear in the output section outsec1 is given by:

1. Section sec1 from file1.o
2. All sections from file2.o, in the order they appear in the file
3. Section sec1 from file3.o, and then section sec2 from file3.o.

If there are any additional input files that contained input sections also named outsec1, these sections are linked following the last section named in the definition of outsec1 (in this example, file3.o (sec2)).

There may be additional input sections in the files file.o, file2.o, and file3.o, other than those specified as going into output section outsec1. These input sections are put into output sections with corresponding names.

Load a Section at a Specified Address. Binding an output section to a specific virtual address is done by a link editor command language option, as shown in the following SECTIONS directive example:

```
SECTIONS {
            outsec addr: { ... }
            etc.
         }
```

where addr is the binding address, expressed as a C language constant. If outsec will not fit at addr (perhaps because of holes in the memory configuration, or because outsec is too large to fit without overlapping some other output section), then **m32ld** issues an appropriate error message.

As long as output sections do not overlap, they can be bound anywhere in configured memory. The SECTIONS directives defining output sections need not be given to **m32ld** in any particular order.

Aligning an Output Section. It is possible to request that an output section be bound to a virtual address that falls on an n-byte boundary, where n is a power of 2. The ALIGN option of the SECTIONS directive performs this function, so that the option

```
ALIGN(n)
```

is equivalent to specifying a binding address of

```
(.+n−1) & ∼(n−1)
```

For example:

```
SECTIONS {
            outsec ALIGN(0x20000): { ... }
            etc.
         }
```

The output section outsec is not bound to any given address, but will be linked to some virtual address that is a multiple of 0x20000 (e.g., at address 0x0, 0x20000, 0x40000, 0x60000, etc.).

Grouping Sections Together. The default allocation algorithm for **m32ld** is:

1. Link all input **.text** sections together into one output section. This output section is called **.text** and is bound at the address 0x100000.
2. Link all input **.data** sections together into one output section. This output section is called **.data** and is bound at an address aligned to 0x8.
3. Link all input **.bss** sections together into one output section. This output section is called **.bss** and is allocated to follow immediately after the output section **.data**. Note that the output section **.bss** is not given any particular address alignment.

Specifying any SECTIONS directive with an ifile inhibits this default allocation.

The default allocation of **m32ld** is equivalent to supplying the following directives:

```
SECTIONS  {
              .text 0x100000: {}
              GROUP ALIGN(0x8):  {
                        .data: {}
                        .bss: {}
                        }
          }
```

The GROUP command ensures that the two output sections **.data** and **.bss** are allocated together (i.e., grouped). Binding or alignment information may be supplied only for the group and not for the output sections contained within the group. The sections making up the group are allocated in the order listed in the directive.

If **.text**, **.data**, and **.bss** are to be placed in the same segment, the following SECTIONS directive could be used:

```
SECTIONS  {
              GROUP:  {
                         .text: {}
                         .data: {}
                         .bss: {}
                      }
          }
```

Note that there are still three output sections (**.text**, **.data**, and **.bss**), but now they are allocated into consecutive virtual memory.

This entire group of output sections could be bound to a starting address, or aligned, simply by adding a field to the GROUP directive. To bind at 0xc0000, use:

```
GROUP 0xc0000: {
```

The output section **.text** is bound at 0xc0000 with this directive. To align to 0x10000 use:

```
GROUP ALIGN(0x10000): {
```

Now the output section **.text** is aligned to 0x10000. In both cases, the remaining members of the group are allocated, in order of their appearance, into the next available memory locations.

When the GROUP directive is not used, each output section is treated as an independent entity. Thus the directive

```
SECTIONS {
            .text: {}
            .data ALIGN(0x20000): {}
            .bss: {}
         }
```

causes the **.text** section to start at virtual address 0x0, and the **.data** section to start at a virtual address aligned to 0x20000. The **.bss** section follows immediately after the **.text** section if there is enough space. If there is not, it follows the **.data** section.

Note: The order in which output sections are defined to the link editor can *not* be used to force a certain allocation order in the output file.

Creating Holes Within Output Sections. The dot symbol (".") can appear in an assignment instruction only within a section definition. When appearing on the left side of an assignment statement, dot causes the link editor location counter to be incremented/reset, and leaves a hole in the output section. Consider the following section definition:

```
outsec: {
            .+= 0x1000;
                        f1.o(.text)
            .+= 0x100;
                        f2.o(.text)
            .= align(4);
                        f3.o(.text)
        }
```

The effects of this command are:

- A 0x1000-byte hole is left at the beginning of the section. Input file f1.o(.text) is linked after this hole.
- The **.text** section of input file f2.o begins 0x100 bytes following the end of f1.o(.text).
- The **.text** section of f3.o is linked to start at the next full word boundary following the text of f2.o, with respect to the beginning of "outsec".

Holes built into output sections in this manner are initialized using a fill character, either the default fill character (0x00) or a supplied fill character. The option **–f** is used to supply a fill character.

To help allocate and align addresses *within* an output section, the link editor treats the output section as if it began at address zero. As a result, if (in the above example), outsec ultimately was linked to start at an odd address, then the part of outsec built from f3.o(.text) would also start at an odd address — even though f3.o(.text) was aligned to a full-word boundary. This could be prevented by specifying an alignment factor for the entire output section:

```
outsec ALIGN(4): {
```

Note that the **m32as** assembler always pads the sections it generates to a full word length, making explicit alignment specifications unnecessary. This also holds true for the **m32cc** compiler.

Expressions that decrement dot are illegal. For example, subtracting a value from the location counter is not allowed, because overwrites are not allowed. The most common operators in expressions that assign a value to dot are += and align.

Creating and Defining Symbols at Link-Edit Time. The assignment instruction of the link editor can be used to give symbols a value that is link-edit dependent. Typically, there are three types of assignments:

1. Use of dot to adjust the **m32ld** location counter during allocation.
2. Use of dot to assign an allocation-dependent value to a symbol.
3. Assigning an allocation-independent value to a symbol.

The first type of assignment has already been discussed in the previous section. The second type provides a means to assign addresses, known only after allocation, to symbols. For example:

```
SECTIONS {
        outsec1: { ... }
        outsec2: {
                                        file1.o
                s2_start =.;
                                        file2.o(s2)
                s2_end =.-1;
            }
    }
```

The symbol s2_start is defined to be the address of file2.o(s2), and s2_end is the address of the last byte of file2.o(s2).

Assignment instructions involving "." must appear within sections definitions, because they are evaluated during *allocation.* Assignment instructions that do not involve ".", although they can appear within sections definitions, typically do not. Such instructions are evaluated after allocation is complete. Reassignment of a defined symbol to a different address is dangerous. If a symbol within **.data** is defined, initialized, and referenced within a set of object files being link edited, the symbol table entry for that symbol will be

changed to reflect the new, reassigned address, but the associated initialized data will not be moved. This link editor issues warning messages for each defined symbol that is being redefined with an *ifile*. However, assignments of absolute values to undefined symbols is safe, because there are no initialized data associated with the symbol.

Allocating a Section Into Named Memory. Within a SECTIONS directive, it is possible to specify that a section be linked somewhere within a *named* memory (as previously shown in a MEMORY directive). This allocation method uses the > notation borrowed from the *UNIX* System concept of "redirected output".

```
MEMORY  {
            mem1:          0=0x000000   1=0x10000
            mem2(RW):      0=0x020000   1=0x40000
            mem3(RW):      0=0x070000   1=0x40000
            mem1:          0=0x120000   1=0x04000
        }
SECTIONS  {
            outsec1:   {f1.o(.data)}  >  mem1
            outsec2:   {f2.o(.data)}  >  mem3
          }
```

This ifile segment directs **m32ld** to place outsec1 anywhere within the memory area named mem1; i.e., somewhere within the address range 0x0-0xffff or 0x120000-0x123fff. Output section outsec2 is placed somewhere in the address range 0x70000-0xaffff.

Initialized Section Holes or BSS Sections. When "holes" are created within a section, the link editor normally outputs bytes of zeros as "fill". By default, **.bss** sections are not initialized at all. That is, no initialized data are generated for any **.bss** section by the assembler nor supplied by the link editor (not even zeros).

Initialization options can be used in a SECTIONS directive to set such holes or output **.bss** sections to an arbitrary two-byte pattern. Such initialization options apply only to **.bss** sections or holes. As an example, an application might want an uninitialized data table to be initialized to a constant value without recompiling the .o files; another application may want a hole in the text area to be filled with a transfer to an error routine.

Either specific areas within an output section or the entire output section may be specified as being initialized. However, since no text is generated for an uninitialized **.bss** section, initializing part of such a section causes the entire section to be initialized. In other words, to combine **a .bss** section with a **.text** or **.data** section (both of which are initialized), or to initialize only part of an output **.bss** section, one of the following must hold:

1. Explicit initialization options must be used to initialize all **.bss** sections in the output section.
2. The link editor must use the default fill value to initialize all **.bss** sections in the output section.

Consider the following ifile:

```
SECTIONS  {
              sec1:  {
                                  f1.o
                                  . =+ 0x200;
                                  f2.o(.text)
                     } = 0x2f2f
              sec2:  {
                                  f1.o(.bss)
                                  f2.o(.bss) = 0x1234
                     }
              sec3:  {
                                  f3.o(.bss)
                                  ...
                     } = 0xffff
              sec4:  {
                                  f4.o(.bss) }
          }
```

In this example, the 0x200 byte hole in section sec1 is filled with WAIT instructions (0x2f2f). In the section sec2, f1.o(.bss) is initialized to the default fill value of 0x00, and f2.o(.bss) is initialized to 0x1234. All **.bss** sections within sec3 as well as holes are initialized to 0xffff. Section sec4 is not initialized (i.e., no data are written to the object file for sec4).

Notes on the Use of m32ld

Notes and special considerations on the use of the link editor, including initialization, use of archive libraries, and other detailed aspects of **m32ld** are presented here.

Changing the Entry Point. By default, a a.out header is written to the output file. The a.out header contains a field for the primary entry point of the file. This field is set by the following rules (listed in the order of application):

1. The value of the symbol in the **−e** flag, if present, is used.
2. The value of the symbol _start, if present, is used.
3. The value of the symbol main, if present, is used.
4. The value zero is used.

Thus, an explicit entry point can be assigned to this m32a.out header field through the **−e** option, or by using an assignment instruction in an input file of the form:

```
_start = expression;
```

If the link editor is called through **m32cc**, a startup routine will automatically be linked in. The user must be careful when calling the link editor directly or when changing the entry

point. The user must supply the startup routine or insure that the program performs the necessary steps in order to execute correctly.

Use of Archive Libraries. Each member of an archive library (e.g., libc.a) is a complete object file (typically consisting of the standard three sections: **.text**, **.data**, and **.bss**). Archive libraries are created through the use of the **m32ar** command from object files generated by running **m32cc** or **m32as**. Each library member has a "magic number". The link editor enforces a policy that all input object files must have the same magic number. Any object file that fails this test is not processed and generates a fatal **m32ld** error. However, this policy has an important exception — members of archive libraries with the wrong magic number are silently skipped. This is not considered an error, and no message is generated.

An archive library is always processed using *selective inclusion.* Only those members which resolve existing undefined symbol references are taken from the library for link editing.

Libraries can be placed both inside and outside section definitions. In both cases, a member of a library is included for linking whenever:

- There is a reference to a symbol defined in that member
- The reference is found by the link editor before the actual scanning of the library.

When a library member is included by searching the library inside a SECTIONS directive, all input sections from the member are included in the output section being defined. When a library member is included by searching the library outside of a SECTIONS directive, all input sections from the member are included into the output section with the same name. That is, the **.text** section of the member goes into the output section named **.text**, the **.data** section of the member goes into **.data**, and the **.bss** section of the member goes into **.bss**. If necessary, **m32ld** defines new output sections to provide a place to put the input sections.

The **−l** option is a shorthand notation for specifying an input file coming from a predefined directory and having a predefined name. By convention, such files are archive libraries, although they need not be. Furthermore, archive libraries can be specified without using the **−l** option, by simply giving the full or relative file pathname.

The ordering of archive libraries is important, since for a member to be extracted from the library it must satisfy a reference that is *known to be unresolved* at the time the library is searched. Archive libraries can be specified more than once; they are searched from the beginning every time they are encountered. The proper order can often be determined with the utility **m32lorder**, as described in **5.5 UTILITIES AND LIBRARY ROUTINES.**

Consider the following example:

- The input files file1.0 and file2.0 each contain a reference to the external function FCN
- Input file1.0 contains a reference to symbol ABC
- Input file2.0 contains a reference to symbol XYZ

- Library liba.a, member 0, contains a definition of XYZ
- Library libc.a, member 0, contains a definition of ABC
- Both libraries have a member 1 that defines FCN.

If the **m32ld** command line is entered as

m32ld file1.o *−l*a file2.o *−l*c

the FCN references are satisfied by liba.a, member 1; ABC is obtained from libc.a, member 0; and XYZ remains undefined because the library liba.a is searched before file 2.o is specified. If the **m32ld** command line is entered as

m32ld file1.o file2.o *−l*a *−l*c

the FCN references are satisfied by liba.a, member 1; ABC is obtained from libc.a, member 0; and XYZ is obtained from liba.a, member 0. If the **m32ld** command line is entered as

m32ld file1.o file2.o *−l*c *−l*a

the FCN references are satisfied by libc.a, member 1; ABC is obtained from libc.a, member 0; and XYZ is obtained from liba.a, member 0. If the **m32ld** command line is entered as

m32ld file.o file2.0 *−l*c *−l*a

The FCN references are satisfied by libc.a., member 1; ABC is obtained from libc.a., member 0; and XYZ is obtained from liba.a., member 0.

The **−u** option can be used to force the linking of library members when the link edit run does not contain an actual external reference to the members. For example,

m32ld *−u* rout1 *−l*a

created the undefined symbol *rout1* in the link editor global symbol table. If any member of library liba.a defines this symbol, that member (and perhaps other members as well) is extracted. Without the **−u** option, there would have been no trigger to cause **m32ld** to search the archive library.

Dealing With Holes in Physical Memory. When memory configurations are defined such that unconfigured areas exist in the virtual memory, each application or user must assume

the responsibility of forming output sections that fit into memory. For example, assume memory is configured as:

```
MEMORY {
        mem1:  o = 0x00000   l = 0x02000
        mem2:  o = 0x40000   l = 0x05000
        mem3:  o = 0x20000   l = 0x10000
    }
```

Let the files f1.o, f2.o, . . . fn.o each contain the standard three sections **.text, .data**, and **.bss**, and suppose the combined **.text** section contains 0x12000 bytes. There is no configured area of memory where this section can be placed. Appropriate directives must be supplied to break up the **.text** output section so that **m32ld** can perform allocation. For example:

```
SECTIONS {
        txt1:  {
                    f1.o (.text)
                    f2.o (.text)
                    f3.o (.text)
                }
        txt2:  {
                    f4.o (.text)
                    f5.o (.text)
                    f6.o (.text)
                }
        etc.
    }
```

Allocation Algorithm. The link editor forms output sections either according to specifications in a SECTIONS directive or by combining input sections with the same name. An output section can contain zero or more input sections. After determining the composition of an output section, **m32ld** must then allocate the necessary configured virtual memory. This task is performed using an algorithm that attempts to minimize fragmentation of memory, thereby increasing the possibility that a link edit run will be able to allocate all output sections within the specified virtual memory configuration. The algorithm proceeds as:

1. Any output sections with explicitly specified binding addresses are allocated.
2. Any output sections to be included in a specific named memory are allocated. In both this and the preceding step, each output section is placed into the *first* available space within the (named) memory, with any alignment taken into consideration.
3. Output sections not handled by one of the above steps are allocated.

If all memory is contiguous and configured (the default case), and no SECTIONS directives are given, then output sections are allocated in the order of appearance to the

link editor, normally **.text**, **.data** and **.bss**. The **.text** output section starts at virtual address 0x100000; the **.data** and **.bss** output sections are grouped together and aligned to a 0x8-byte virtual address. Otherwise, output sections are allocated in the order they were defined or made known to the link editor. The first available space large enough to hold them is used.

Subsystems (Incremental) Link Editing. To help generate a large system with modular and hierarchical design methodology, **m32ld** provides the capability to form subsystems and link edit in smaller, more manageable increments. As previously mentioned, the output of the link editor can be used as an input file to subsequent **m32ld** runs providing that the relocation information is retained (**−r** option). In large applications it may be desirable to partition C programs into subsystems, link each subsystem independently, and then link-edit the entire application. For example:

Step 1.

```
m32ld −r −o outfile1 ifile1
/* ifile */
SECTIONS  {
          ss1:    {
                          f1.o
                          f2.o
                          ...
                          fn.o
                  }
          }
```

Step 2.

```
m32ld −r −o outfile2 ifile2
/* ifile2 */
SECTIONS  {
          ss2:    {
                          g1.o
                          g2.o
                          ...
                          gn.o
                  }
          }
```

Step 3.

```
m32ld −a −m −o final.out outfile1 outfile2
```

By judiciously forming subsystems, applications may achieve a form of "incremental link editing" whereby it is only necessary to re-link a portion of the total link edit when a few programs are recompiled.

Two simple rules are followed when applying this technique:

1. Place SECTIONS declarations only in incremental link edits. Be concerned only with the formation of output sections from input files and input sections. Do not bind output sections in these runs.
2. Only allocation and memory directives, as well as any assignment statements, should be included in the final **m32ld** call.

Nonrelocatable Input Files. Normally an input file produced by a previous **m32ld** run was produced under the **−r** option. This option preserves relocation information and permits sections of the output file to be relocated by subsequent **m32ld** runs.

Upon detecting an input file that does not have relocation or symbol table information, the link editor issues a warning message. Such information may have been removed by the **−a** or **-s** options of the link editor, as described in **5.3.1 Command Line Options**. However, the link editor continues using the nonrelocatable input file.

For such a link edit to succeed (i.e., to actually and correctly incorporate all input files, relocate all symbols, resolve all unresolved references, etc.), two conditions must be met by the nonrelocatable input file:

1. Each input file must not contain any unresolved external references.
2. Each input file must be bound at the same virtual address where it was bound during the **m32ld** run that created it.

Note: The **m32ld** link editor does not issue an error message if these two conditions are not met for all nonrelocatable input files. Therefore, extreme care must be exercised when supplying such input files to the link editor.

DSECT, COPY and NOLOAD Sections

Sections may be given a "type" in a section definition as shown in the following example:

```
SECTIONS
{
        name1 0x200000 (DSECT)     :{file1.o}
        name2 0x400000 (COPY)      :{file2.o}
        name3 0x600000 (NOLOAD)    :{file3.o}
}
```

The DSECT option creates what is called a "dummy section". A "dummy section" has the following properties:

- It does not participate in the memory allocation for output sections. As a result, it takes up no memory and does not show up in the memory map (the "−m" option) generated by the link editor.
- It may overlay other output sections and even unconfigured memory. DSECTs may overlay other DSECTs.

- The global symbols defined with the "dummy section" are relocated normally. That is, they appear in the output file's symbol table with the same value they would have had if the DSECT were actually loaded at its virtual address. DSECT-defined symbols can be referenced by other input sections. Undefined external symbols found within a DSECT will cause specified archive libraries to be searched and any members which define such symbols will be link-edited normally (i.e., not in the DSECT or as a DSECT).
- None of the section contents, relocation information, or line number information associated with the section is written to the output file.

In the above example, none of the sections from file1.o are allocated, but all symbols are relocated as though the sections were link edited at the specified address. Other sections may refer to any of the global symbols and they are resolved correctly.

A "copy section" created by the COPY option is similar to a "dummy section"; the only difference being that the contents of a "copy section" and all associated information is written to the output file.

A section with the "type" of NOLOAD differs in only one respect from a normal output section: its text and/or data is not written to the output file. A NOLOAD section is allocated virtual space, appears in the memory map, etc.

Output File Blocking

The BLOCK option, which can be applied to any output section or GROUP directive, is used to direct *m32ld* to align a section at a specified byte offset in the output file. It has no effect on the address at which the section is allocated, nor on any part of the link editor process. It is used only to adjust the physical position of the section in the output file.

```
SECTIONS
{
      .text BLOCK(0x200) : { }
      .data ALIGN (0x20000) BLOCK(0x200) : { }
}
```

With this SECTIONS directive, *m32ld* will ensure that each section (.text and .data) is physically written at a file offset which is a multiple of 0x200 (e.g., at an offset of 0, 0x200, 0x400,..., etc. in the file).

5.3.3 Error Messages

Corrupt Input Files

The following error messages indicate that the input file is corrupt, nonexistent, or unreadable. The user should check that the file is in the correct directory with the correct permissions. If the object file is corrupt, try recompiling or reassembling it.

- Can't open *name*
- Can't read archive header from archive *name*

- Can read file header of archive *name*
- Can't read 1st word of file *name*
- Can't seek to the beginning of file *name*
- Fail to read file header of *name*
- Fail to read lnno of section sect of file *name*
- Fail to read magic number of file *name*
- Fail to read section headers of file *name*
- Fail to read section headers of library *name* member *number*
- Fail to read symbol table of file *name*
- Fail to ready symbol table when searching libraries
- Fail to read the aux entry of file *name*
- Fail to read the field to be relocated
- Fail to seek to symbol table of file *name*
- Fail to seek to symbol table when searching libraries
- Fail to seek to the end of library *name* member *number*
- Fail to skip aux entries when searching libraries
- Fail to skip the mem of struct of *name*
- Illegal relocation type
- Line nbr entry (*num num*) found for non-relocatable symbol: section *sect*, file *name*
- No reloc entry found for symbol
- Reloc entries out of order in section *sect* of file *name*
- Seek to *name* section *sect* failed
- Seek to *name* section *sect* lnno failed
- Seek to *name* section *sect* reloc entries failed
- Seek to relocation entries for section *sect* in file *name* failed

Errors During Output

These errors and messages occur because the link editor cannot write to the output file, usually indicating that the file system is out of space.

- Cannot complete output file *name*. Write error.
- Fail to copy the rest of section *num* of file *name*
- Fail to copy the bytes that need no reloc of section *num* of file *name*
- I/O error on output file *name*

Internal Errors

These messages indicate that something is wrong with the link editor internally. There is probably nothing the user can do except get help.

- Attempt to free nonallocated memory
- Attempt to reinitialize the SDP aux space
- Attempt to reinitialize the SDP slot space
- Default allocation didn't put .data and .bss into the same region
- Failed to close SDP symbol space
- Failure dumping an AIDFNxxx data structure
- Failure in closing SDP aux space
- Failure to initialize the SDP aux space
- Failure to initialize the SDP slot space
- Internal error: audit_groups, address mismatch
- Internal error: audit_groups finds anode failure
- Internal error: audit_regions detected no regions built
- Internal error: fail to seek to the member of *name*
- Internal error: in allocate lists, list confusion (*num num*)
- Internal error: invalid aux table id
- Internal error: invalid symbol table id
- Internal error: negative aux table id
- Internal error: negative symbol table id
- Internal error: no symtab entry for DOT
- Internal error: out of tv slots
- Internal error: split_scns, size of *sect* exceeds its new displacement
- Internal error: .tv not aligned
- Internal error: .tv not built

Allocation Errors

These error messages appear during the allocation phase of the link edit. They generally appear if a section or group will not fit at a certain address, or if the given MEMORY, REGION, or SECTION directives conflict in some way. If using an ifile, the user should check that MEMORY and SECTION directives allow enough room for the sections, that nothing overlaps, and that nothing is being placed in configured memory.

- Bond address *address* for *sect* is not in configured memory

- Bond address *address* for *sect* overlays previously allocated section *sect* at *address*
- Can't allocate output section *sect*, of size *num*
- Can't allocate section *sect* into owner *mem*
- Default allocation failed: *name* is too large
- GROUP containing section *sect* is too big
- Memory types *name1* and *name2* overlap
- Output section *sect* not allocated into a region
- *Sect* at *address* overlays previously allocated section *sect* at *address*
- *Sect*, bonded at *address*, won't fit into configured memory
- *Sect* enters unconfigured memory at *address*
- Section *sect* in file *name* is too big

Misuse of Link Editor Directives

These errors and messages arise from the misuse of an input directive. Review the appropriate section in the manual.

- Adding *name*(*sect*) to multiple output sections
  The input section is mentioned twice in the SECTION directive.
- Bad attribute value in MEMORY directive: *c*
  An attribute must be one of "R", "W", "X", or "I".
- Bad flag value in SECTIONS directive, *option*
  Only the "−l" option is allowed inside of a SECTIONS directive.
- Bad fill value
  The fill value must be a two-byte constant.
- Bonding excludes alignment
  The section will be bound at the given address, regardless of the alignment of that address.
- Cannot align a section within a group
- Cannot bond a section within a group
- Cannot specify an owner for sections within a group
  The entire group is treated as one unit, so the group may be aligned or bound to an address, but the sections making up the group may not be handled individually.
- DSECT *sect* can't be given an owner
- DSECT *sect* can't be linked to an attribute
  Since dummy sections do not participate in the memory allocation, it is meaningless for a dummy section to be given an owner or an attribute.
- REGIONS command not allowed in any instantiation other than b16

- *Sect* is a reserved section name
  Currently only ".tv" is a reserved section name.
- Section *sect* not built
  The most likely cause of this is a syntax error in the SECTIONS directive.
- Semicolon required after expression
- Statement ignored
  Caused by a syntax error in an expression.
- Usage of unimplemented syntax

Misuse of Expressions

These errors and messages arise from the misuse of expressions.

- Absolute symbol *name* being redefined
  An absolute symbol may not be redefined.
- ALIGN illegal in this context
  Alignment of a symbol may only be done within a SECTIONS directive.
- Attempt to decrement DOT
- Illegal operator in expression
- Misuse of DOT symbol in assignment instruction
  The DOT symbol (".") cannot be used in assignment statements that are outside SECTIONS directives.
- Symbol *name* is undefined
  All symbols referenced in an assignment statement must be defined.
- Symbol *name* from file *name* being redefined
  A defined symbol may not be redefined in an assignment statement.
- Undefined symbol in expression

Misuse of Options

These errors and messages arise from the misuse of options.

- Both −r and −s flags are set. −s flag turn off
  Further relocation requires a symbol table
- Can't find library libx.a
- −L path too long (*string*)
- −o file name too large (>128 char), truncated to (*string*)
- Too many −L options, 7 allowed

Some options require whitespace before the argument, some do not. Including extra whitespace, or not including the required whitespace is the most likely cause of the following messages.

- *option* flag does not specify a number
- *option* is an invalid flag
- −e flag does not specify a legal symbol name *name*
- −f flag does not specify a two-byte number
- No directory given with −L
- −o flag does not specify a valid file name: *string*
- the −l flag (specifying a default library) is not supported
- −u flag does not specify a legal symbol name: *name*

Space Restraints

The following error messages may occur if the link editor attempts to allocate more space than is available. This is more likely to occur on a *PDP* 11/70 Computer than on other machines. The user should attempt to decrease the amount of space used by the link editor. This may be accomplished by making the ifile less complicated, or by using the "−r" option to create intermediate files.

- Fail to allocate *num* bytes for slotvec table
- Internal error: aux table overflow
- Internal error: symbol table overflow
- Memory allocation failure on *num*-byte 'calloc' call
- Memory allocation failure on realloc call
- Run is too large and complex

Miscellaneous Errors

These errors and messages occur because of a misuse of the link editor in general.

- Archive symbol table is empty in archive *name*, execute 'ar ts *name*' to restore archive symbol table

  On systems with a random access archive capability, the link editor requires that all archives have a symbol table. This symbol table may have been removed by strip.

- Can't create intermediate id file *name*
- Can't open internal file *name*

  These two messages are possible only when the link editor is two processes. This would indicate that the temp directory (usually /tmp or /usr/tmp) is out of space, or that the link editor does not have permission to write in it.

- Can't create output file *name*
  The user may not have write permission in the directory where the output file is to be written.

- Failure to load pass 2 of ld
  This can only occur when the link editor is built as two processes (i.e., on the PDP 11/70). The most likely cause is that the second process is not accessible to the first one.
- File *name* has a section name which is a reserved ld identifier: .tv
- File *name* has no relocation information
- File *name* is of unknown type, magic number = *num*
- Ifile nesting limit exceeded with file *name*
  Ifiles may be nested 16 deep.
- Library *name*, member has no relocation information
- Multiply defined symbol *sym*, in *name* has more than one size
  A multiply defined symbol may not have been defined in the same manner in all files.
- *name(sect)* not found
  An input section specified in a SECTIONS directive was not found in the input file.
- Section *sect* starts on an odd byte boundary!
  This will happen only if the user specifically binds a section at an odd boundary.
- Sections .text .data or .bss not found. Optional header may be useless
  The *UNIX* System a.out header uses values found in the .text, .data, and .bss section headers.
- Undefined symbol *sym* first referenced in file *name*
  Unless the −r option is used, the link editor requires that all referenced symbols must be defined.
- Unexpected EOF
  Syntax error in the ifile.

5.3.4 Syntax Diagram for Input Directives

```
<ifile>          −> {<cmd>}

<cmd>            −> <memory>
                 −> <sections>
                 −> <assignment>
                 −> <filename>
                 −> <options>
<memory>         −> MEMORY {<memory_spec>{[,]<memory_spec>}}

<memory_spec>    −> <name>[<attributes>]:<origin_spec>i,]<length_spec>

<attributes>     −> ({R|W|X|I})
```

```
<origin_spec>     -> <origin> = <long>

<length_spec>     -> <length> = <long>

<origin>          -> ORIGIN |o|org|origin

<length>          -> LENGTH|l|len|length

<sections>        -> SECTIONS {{<sec_or_group>}}

<sec_or_group>    -> <section>|<group>|<library>

<group>           -> GROUP <group_options>: {<section_list>}[<mem_spec>]

<section_list>    -> <section>{[,]<section>}

<section>         -> <name> <sec_options>:{<statement_list>}[<fill>][<mem_spec>]

<group_options>   -> [<addr>][<align_option>][<block_option>]

<sec_options>     -> [<addr>][<align_option>][<block_option>][<type_option>]

<addr>            -> <long>

<align_option>    -> <align>(<long>)

<align>           -> ALIGN|align

<block_option>    -> <block>(<long>)

<block>           -> BLOCK|block

<type_option>     -> (DSECT)|(NOLOAD|(COPY)

<fill>            -> = <long>

<mem_spec>        -> > <name>
                  -> > <attributes>

<statement>       -> <filename>[(<name_list>)][<fill>]
                  -> <library>
                  -> <assignment>

<name_list>       -> <name>{[,]<name>}

<library>         -> -|<name>
```

```
<assignment>    -> <lside><assign_op><expr><end>

<lside>         -> <name>|.

<assign_op>     -> =|+=|-=|*=|/=

<end>           -> ;|,

<expr>          -> <expr><binary_op><expr>
                -> <term>

<binary_op>     -> *|/|%
                -> +|-
                -> >>|<<
                -> ==|!=|>|<|<=|>=
                -> &
                -> |
                -> &&
                -> ||

<term>          -> <long>
                -> <name>
                -> <align>(<term>)
                -> (<expr)
                -> <unary_op><term>

<unary_op>      -> !|-

<options>       -> -a
                -> -e<wht_space><name>
                -> -f<wht_space><long>
                -> -i
                -> -l<name>
                -> -m
                -> -o<wht_space><filename>
                -> -r
                -> -s
                -> -t
                -> -u<wht_space><name>
                -> -x
                -> -z
                -> -F
                -> -L<pathname>
                -> -M
                -> -N
                -> -S
```

	−>	−V
	−>	−VS
	−>	−X
<name>	−>	any valid symbol name
<long>	−>	any valid long integer constant
<wht_space>	−>	blanks, tabs and newlines
<filename>	−>	any valid *UNIX* System filename. This may include a full or partial pathname
<pathname>	−>	any valid *UNIX* System pathname (full or partial)

5.4 OBJECT FILE FORMAT

The output file produced by the **m32as** assembler and the **m32ld** link editor is in a format called the Common Object File Format (COFF). Several target machines use this format, and more than one operating system on some of those machines use it. Hence the word Common is both descriptive and widely recognized as a unique name. Because systems other than the *WE* 32100 Microprocessor use COFF, the format includes some symbols and fields that appear to be extraneous. These items are extraneous for processor users, and are only included to maintain commonality.

An object file that contains no errors or unresolved references can be executed on the target processor.

The object file supports user-defined sections and contains extensive information for symbolic software testing. An object file consists of a file header, optional header information, a table of section headers, the data corresponding to the section headers, relocation information, line numbers, a symbol table, and a string table. Figure 5-6 shows this overall structure.

The common object file is not only simple enough to be incorporated into existing projects, but advanced enough to meet the needs of yet unspecified operating systems. Some key features are:

- Applications may add system-dependent information to the object file without causing access utilities to become obsolete.
- A wealth of symbolic information is provided for the use of debuggers and other applications.
- Some modifications in object file construction may be made by the user or source file application at compile time.

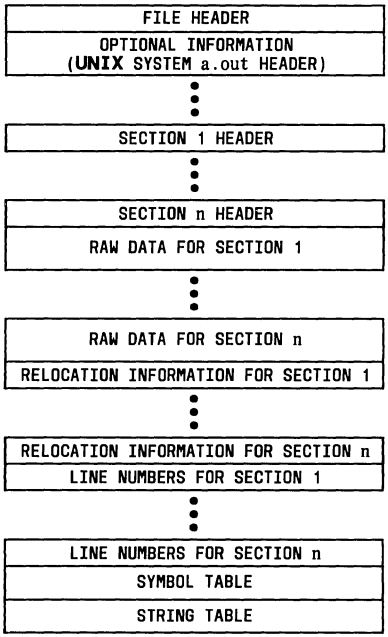


Figure 5-6. Object File Format

5.4.1 Definitions

The object file specification uses the following terms:

Section	A section is the smallest portion of an object file that can be relocated and can be treated as one distinct entity. The default case has three sections: **.text, .data**, and **.bss**. Additional sections are added to the default sections for multiple text segments, shared data segments, or user-specified segments.
Physical Address	This is the 32-bit offset of a section with respect to the beginning of memory. All relocatable references in a section assume that the section resides at that address at execution time.
Virtual Address	The virtual address is used by only a few systems. In *WE* 32100 Microprocessor object files, the physical address is equivalent to the virtual address.

5.4.2 File Header

The file header contains the twenty bytes of information described in Table 5-8. The last two bytes are flags that may be of use to **m32ld**. The manual page for FILEHDR, found in **5.6 SGP MANUAL PAGES**, gives the exact C language structure for the file header.

The size of optional header information should be used by all referencing programs that need to seek the beginning of section header table.

Table 5-8. File Header Contents

Bytes	Contents	Mnemonic	Description
0—1	Unsigned Short	f_magic	Magic number equal to 0560, also defined by the mnemonic FBOMAGIC.
2—3	Unsigned Short	f_nscns	Number of section headers (equals the number of sections).
4—7	Long Int	f_timdat	Time and date stamp containing the number of elapsed seconds since 00:00:00 GMT, January 1, 1970.
8—11	Long Int	f_symptr	File pointer containing the starting address of the symbol table.
12—15	Long Int	f_nsyms	Number of entries in the symbol table.
16—17	Unsigned Short	f_opthdr	Number of bytes in the optional header.
18—19	Unsigned Short	f_flags	Flags (see Table 5-9).

Flags

The last two bytes of the file header are flags that describe the type of the object file. The *WE* 32100 Microprocessor version of the COFF has no use for some of these flags, but keeps them to maintain commonality.

The notation AR16WR in Table 5-9 signifies the architecture of the host machine where the file was created. The AR stands for architecture, the digits give the number of bits per word, W signifies left-to-right byte-ordering (most significant bit first), and WR signifies right-to-left byte-ordering (least significant bit first). The AR32W machines are either members of the *AT&T* 3B Computer family or the "MAXI" version of the *UNIX* Operating System that runs on a host *IBM* Computer.

Table 5-9. File Header Flags		
Mnemonic	**Flag**	**Meaning**
F_RELFLG	00001	Relocation information stripped from file.
F_EXC	00002	File is executable (i.e., no unresolved external references).
F_LNNO	00004	Line numbers stripped from file.
F_LSYMS	00010	Local symbols stripped from file.
F_MINMAL	00020	Not applicable to the *WE* 32100 Microprocessor.
F_SWABD	00100	This file has had its bytes swapped (i.e., the bytes of symbol table name entries have been reversed.)
F_AR16WR	00200	Created on AR16WR machine (e.g., *PDP* 11/70 Computer).
F_AR32WR	00400	Created on AR32WR machine (e.g., *VAX* 11/780 Computer).
F_AR32W	01000	Created on AR32W machine (e.g., 3B MAXI Computer).
F_PATCH	02000	Not applicable to the *WE* 32100 Microprocessor.
F_BM32B	020000	File contains *WE* 32100 Microprocessor code

Optional Header Information. The template for optional information varies among different systems that use the COFF. Applications place all systems-dependent information into this record. General utility programs (e.g., the table access library functions, the **m32size** utility, the **m32strip** utility, etc.) can be made to work properly on any Common object file by seeking past this record using the size of optional header information in the file header (bytes 16 and 17).

Standard *UNIX* System a.out Header

By default, files produced by the link editor always have a standard *UNIX* Operating System **a.out** header in the optional header field. It contains the 28 bytes of information listed in Table 5-10.

Table 5-10. Optional Header Contents		
Bytes	**Name**	**Contents**
0—1	magic	Magic number
0—3	vstamp	Version stamp
4—7	tsize	Size of text (bytes)
8—11	dsize	Size of initialized data (bytes)
12—15	bsize	Size of uninitialized data (bytes)
16—19	entry	Entry point
20—23	text_start	Base address of text
24—27	data_start	Base address of data

Possible values for the *UNIX* System header magic number are 0407, 0410 and 0413.

The following C language *struct* declaration is currently used for standard *UNIX* Operating System **a.out** file header:

```
typedef struct aouthdr {
        short   magic;
        short   vstamp;
        long    tsize;          /* text size in bytes,padded to FW
                                   bdry*/
        long    dsize;          /* initialized data " "*/
        long    bsize;          /* uninitialized data " "*/
        long    entry;          /* entry pt.*/
        long    text_start;     /* base of text used for this file*/
        long    data_start;     /* base of data used for this file*/
} AOUTHDR;
```

5.4.3 Section Header Table

Every object file has a section header table that specifies the layout of data within the file. Each section within an object file also has its own header.

The section header table consists of one entry for every section in the file. Each entry contains the information in Table 5-11.

Table 5-11. Section Header Contents

Bytes	Name	Contents
0—7	S_name	8-character null padded section name
8—11	S_paddr	Physical address of section
12—15	S_vaddar	Virtual address of section
16—19	S_size	Section size
20—23	S_scnptr	File pointer to raw data
24—27	S_relptr	File pointer to relocation entries
28—31	S_lnnoptr	File pointer to line number entries
32—33	S_nreloc	Number of relocation entries
34—35	S_nlnno	Number of line number entries
36—39	S_flags	Flags. Only byte 36 is used; bytes 37—39 are pads.

Section sizes are always padded to a multiple of 4 bytes.

File pointers are byte offsets that can be used to directly and exactly locate the start of data, relocation, or line number entries for the section. They can be readily used with the *UNIX* Operating System function **fseek**(3S).

Flags

The flag field indicates section types. The flags are defined in Table 5-12.

Table 5-12. Section Types

Mnemonic	Flag	Meaning
STYP_REG	0x00	Regular section (allocated, relocated, loaded)
STYP_DSECT	0x01	Dummy section (not allocated, relocated, not loaded)
STYP_NOLOAD	0x02	Noload section (allocated, relocated, not loaded)
STYP_GROUP	0x04	Grouped section (formed from input sections)
STYP_PAD	0x08	Padding section (not allocated, not relocated, loaded)
STYP_COPY	0x10	Copy section (for a decision function used in updating fields; not allocated, not relocated, loaded, relocation and line number entries processed normally)
STYP_TEXT	0x20	Section contains executable text.
STYP_DATA	0x40	Section contains initialized data.
STYP_BSS	0x80	Section contains uninitialized data.

The C language data structure that is used to declare section headers can be found on the manual page for SCNHDR.H(5L) in **5.6 SGP MANUAL PAGES**.

.bss Section Header

The one anomaly in the section header table is the entry for uninitialized data in a **.bss** section. A **.bss** section has a size, symbols that refer to it, and symbols that are defined in it. At the same time a **.bss** section has no relocation entries, no line number entries, and no data. Therefore, a **.bss** section has an entry in the section header table but occupies no space in the section area of the file. That is, there are no raw data for **.bss** sections in the area of the COFF immediately following the section headers. In this case, the number of relocation and line number entries, as well as all file pointers in a **.bss** section header, are zero.

5.4.4 Sections

Figure 5-6 shows that section headers are followed by the appropriate number of bytes of text or data. The raw data for each section begins on a full word boundary in the file.

Files produced by the compiler and the assembler always contain three sections, **.text**, **.data**, and **.bss**. The **.text** section contains the instruction text (e.g., code), the **.data** section contains initialized data variables, and the **.bss** section contains uninitialized data variables.

The link editor SECTIONS directives allow users to describe how input sections are to be combined, to direct where to place output sections, and to rename output sections. If no SECTIONS directives are given, each input section appears in an output section of the same name. For example, if a number of object files from the compiler are linked, each containing the three sections **.text, .data**, and **.bss**, the output object file will also contain three sections, **.text, .data**, and **.bss**.

5.4.5 Relocation Information

Object files have one relocation entry for each relocatable reference in the text or data. The relocation information consists of entries with the following 10-byte format:

VIRTUAL ADDRESS	4 BYTES
SYMBOL TABLE INDEX	4 BYTES
RELOCATION TYPE	2 BYTES

The first 4 bytes of the entry is the virtual address of the text or data. The next 4 byte field, counted from 0, indexes the symbol table entry being referenced. The last 2 bytes indicate the type of relocation to be applied.

The C language data structure that is used to declare relocation information can be found on the manual page for RELOC in **5.6 SGP MANUAL PAGES.**

As the link editor reads each input section and performs relocation, the relocation entries are read. Relocation entries direct how references found within the input section are treated.

Relocation types currently recognized are:

R_ABS The reference is absolute and no relocation is needed. The entry is ignored.

R_DIR32 The entry is a direct, 32-bit reference to the virtual address of the symbol.

R_DIR32S The entry is a direct, 32-bit reference to the virtual address of the symbol, with the 32-bit value stored in reverse order in the object file.

The **m32cc** compiler and **m32as** assembler automatically generate relocation entries, which are automatically used by the link editor. The **−r** link editor option retains relocation entries in an object file. The **−a** link editor option is used to remove relocation entries from an object file.

5.4.6 Line Numbers

The **m32cc** compiler generates an entry in the object file for every C language source line where a breakpoint can be inserted. Users can then reference line numbers when using the appropriate debugger. All line numbers in a section are grouped by function, as shown below.

SYMBOL INDEX	0
PHYSICAL ADDRESS	LINE NUMBER
PHYSICAL ADDRESS	LINE NUMBER
.	.
.	.
.	.
SYMBOL INDEX	0
PHYSICAL ADDRESS	LINE NUMBER
PHYSICAL ADDRESS	LINE NUMBER

The first entry in a function grouping has line number zero, and has an index into the symbol table for the entry containing the function name in place of the physical address. Subsequent entries will have actual line numbers and addresses of the text corresponding to the line numbers. The line number entries appear in increasing order of address.

The C language data structure that is used to declare line numbers can be found on the manual page for LINENUM in **5.6 SGP MANUAL PAGES.**

5.4.7 Symbol Table

The ordering of symbols in the symbol table determines the scope of the symbols. The order of symbols in the symbol table is, therefore, very important because of the symbolic debugging requirements for the SGP. Symbols appear in the sequence shown on Figure 5-7.

The word STATICS on Figure 5-7 refers to symbols defined in the C language storage class *static* outside any function. The symbol table consists of at least one fixed-length entry per symbol, with some symbols followed by an auxiliary entry of the same size. The entry for each symbol is a structure that holds the value, the type and other information.

Special Symbols

The symbol table contains some special symbols that are created by the compiler, assembler, link editor, or utilities. These symbols are listed in Table 5-13.

When a structure, union, or enumeration has no tag name (a legitimate C language syntax) the symbol table must create a name. The name chosen by the symbol table is **.xfake**, where **x** is an integer. If there are 3 unnamed structures, unions, or enumerations in the source; their tag names will be **.0fake**, **.1fake**, and **.2fake**.

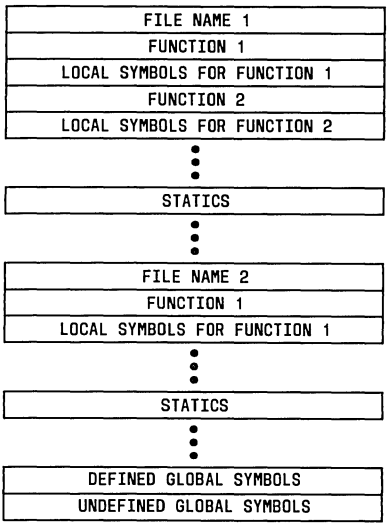


Figure 5-7. COFF Symbol Table

Table 5-13. Special Symbols in the Symbol Table

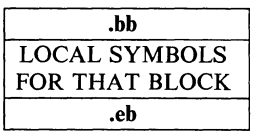
Symbol	Meaning
.file	File name
.text **.data** **.bss**	Address of **.text** section Address of **.data** section Address of **.bss** section
.bb **.eb**	Address of start of inner block Address of end of inner block
.bf **.ef**	Address of start of function Address of end of function
.target	Pointer to the function returned structure or union.
.xfake	Dummy tag name for structure, union, or enumeration.
.eos	End of members of structure, union, or enumeration.
.etext	Next available address after the end of the **.text** section.
.edata	Next available address after the end of the **.data** section.
.end	Next available address after the end of the **.bss** section.

Six of these special symbols occur in pairs. The **.bb** and **.eb** symbols indicate the boundaries of inner blocks; a **.bf** and **.ef** pair brackets each function; and a **.xfake** and **.eos** pair names and defines the limit of structures, unions, and enumerations that were not named. The **.eos** symbol also appears after named structures, unions, and enumerations.

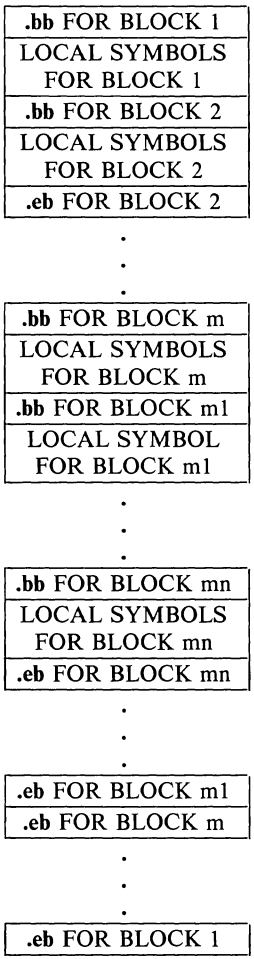
Each of the special symbols has different information stored in the symbol table entry as well as in the auxiliary entry.

Inner Blocks. The C language defines a block as a compound statement that begins and ends with braces ({ and }). An inner block is a block that occurs within a function (which is also a block).

For each inner block that has local symbols defined, a special symbol **.bb** is put in the symbol table immediately before the first local symbol of that block. Also a special symbol **.eb** is put in the symbol table immediately after the last local symbol of that block. The sequence is:



Because inner blocks can be nested by several levels, nested inner blocks may have the following sequence:



For example: given the following code:

```
{                               /* Begin Block 1 */
   int i;
   char c;
   . . .
   {                            /* Begin Block 2 */
            long a;
            . . .
            {                   /* Begin Block 3 */
                     int x;
                     . . .
            }                   /* End Block 3 */
   }                            /* End Block 2 */
   {                            /* Begin Block 4 */
            long i;
            . . .
   }                            /* End Block 4 */
}                               /* End Block 1 */
```

The symbol table would look like:

.bb for Block 1
i c
.bb for Block 2
a
.bb for Block 3
x
.eb for Block 3
.eb for Block 2
.bb for Block 4
i
.eb for Block 4
.eb for Block 1

Symbols for Functions. For each function, a special symbol **.bf** is put between the function name and the first local symbol of the function in the symbol table. Also a special symbol **.ef** is put immediately after the last local symbol of the function in the symbol table. The sequence is:

FUNCTION NAME
.bf
LOCAL SYMBOLS
.ef

If the return value of the function is a structure or union, the special **.target** symbol is put between the function name and the **.bf**. The sequence becomes:

FUNCTION NAME
.target
.bf
LOCAL SYMBOLS
.ef

The **m32cc** compiler creates **.target** to store the function-returned structure or union. The **.target** symbol is an automatic variable with pointer type. Its stack offset (value field in the symbol table entry) is always zero.

Symbol Table Entries

All symbols, regardless of storage class and type, have the same format for their entries in the symbol table. The symbol table entries contain the 18 bytes of information shown in Table 5-14. Indices for symbol table entries begin at zero and count upward. Each auxiliary entry also counts as one symbol.

Table 5-14. Symbol Table Entry Format

Bytes	Mnemonic	Contents
0—7	n_name	These 8 bytes contain either the name or a pointer to the name of the symbol
8—11	n_value	Value (depends on storage class)
12—13	n_scnum	Section number
14—15	n_type	Type specification (basic and derived types)
16	n_sclass	Storage class
17	n_numaux	Number of auxiliary entries

Symbol Name Field (n_name). The first eight bytes in the symbol table entry are a union of a character array and two long integers. These eight bytes are described in Table 5-15. If the symbol name is eight characters or less, the (null-padded) symbol name will be stored in **n_name** field. If the symbol name is longer than eight characters, then the entire symbol name will be stored in the string table. In this case, the eight bytes will contain two long integers, the first of which will be zero, and the second will be the offset (relative to the beginning of the string table) of the name in the string table. Since there can be no symbols with a null name, the zeroes in the first four bytes serve to distinguish a symbol table entry with an offset from one with a name in the first eight bytes.

Table 5-15. n_name Entry Formats

Bytes	Mnemonic	Description
0—7	n_name	Eight character null-padded symbol name.
0—3	n_zeroes	Zero in this field indicates the symbol name is in the string table.
4—7	n_offset	Offset of the symbol name in the string table.

Some special symbols are created by the compiler and link editor, as discussed in **Special Symbols**. The names of special symbols usually start with a dot (.); e.g., **.file**, **.5fake**, **.bb**.

Symbol Value Field and Storage Classes (n_value). The meaning of the value of a symbol depends on its storage class. Table 5-16 lists storage classes, values, and meanings.

Table 5-16. Symbol Values

Storage Class	Mnemonic	Storage Class Value	Meaning of Value Field
Automatic variable	C_AUTO	1	Stack offset (bytes)
External symbol	C_EXT	2	Relocatable address
Static	C_STAT	3	Relocatable address
Register variable	C_REG	4	Register number
Label	C_LABEL	6	Relocatable address
Member of structure	C_MOS	8	Offset (bytes)
Function argument	C_ARG	9	Stack offset
Structure tag	C_STRTAG	10	0 (always zero)
Member of union	C_MOU	11	Offset (bytes)
Union tag	C_UNTAG	12	0
Type definition	C_TPDEF	13	0
Enumeration tag	C_ENTAG	15	0
Member of enumeration	C_MOE	16	Enumeration value
Register parameter	C_REGPARM	17	Register number
Bit field	C_FIELD	18	Bit displacement
Beginning and end of block	C_BLOCK	100	Relocatable address
Beginning and end of function	C-FCN	101	Relocatable address
End of structure	C_EOS	102	Size
File name	C_FILE	103	(See Note)
Duplicated tag	C_ALIAS	105	Tag index

Note: If the current symbol is the last symbol that has storage class C_FILE (**.file** symbol), its value is the symbol table entry index of the first global symbol. Otherwise the symbol value equals the symbol table entry index of the next **.file** symbol (i.e., the **.file** entries form a one-way linked list in the symbol table).

Relocatable symbols have a value equal to the virtual address of that symbol. When the section is relocated by the link editor, the value of these symbols changes.

The **m32cprs** symbol table compressor utility creates the C_ALIAS mnemonic. This utility, which is described in **5.5 UTILITIES AND LIBRARY ROUTINES**, removes duplicated structure, union, and enumeration definitions and puts ALIAS entries in their places.

There are also some dummy storage classes defined in the header file. They are used only internally by the compiler and the assembler. These storage classes are listed in Table 5-17.

Table 5-17. Dummy Storage Classes		
Storage Class	**Mnemonic**	**Value**
Physical end of function	C_EFCN	−1
External definition	C_EXTDEF	5
Undefined label	C_ULABEL	7
Uninitialized static	C_USTATIC	14
Used only by utility programs	C_LINE	104

Table 5-18 lists special symbols that are restricted to certain storage classes.

Table 5-18. Restricted Special Symbols	
Special Symbol	**Storage Class**
.file	C_FILE
.bb	C_BLOCK
.eb	C_BLOCK
.bf	C_FCN
.ef	C_FCN
.target	C_AUTO
.xfake	C_STRTAG,C_UNTAG,C_ENTAG
.eos	C_EOS
.text	C_STAT
.data	C_STAT
.bss	C_STAT

Some storage classes are used only for certain special symbols. Table 5-19 summarizes these storage classes.

Table 5-19. Restricted Storage Classes	
Storage Class	**Special Symbol**
C_BLOCK	**.bb,.eb**
C_FCN	**.bf,.ef**
C_ESO	**.eos**
C_FILE	**.file**

Section Number Field (n_scnum). Table 5-20 lists the section numbers and their meanings.

Table 5-20. Section Numbers

Section Number	Mnemonic	Meaning
−2	N_DEBUG	Special symbolic debugging symbol
−1	N_ARB	Absolute symbol
0	N_UNDEF	Undefined external symbol
1−077767	N_SCNUM	Section number where symbol was defined

A special section number (−2) marks symbolic debugging symbols, including structure/union/enumeration tag names, typedefs, and the name of the file. A section number of −1 indicates that the symbol has a value, but is not relocatable. Examples of absolute-valued symbols include automatic and register variables, function arguments, and **.eos** symbols. If the SECTIONS directive-capability of the link editor is not used, **.text**, **.data**, and **.bss** symbols default to section numbers 1, 2, and 3, respectively.

A section number of zero indicates a relocatable external symbol that is not defined in the current file, with one exception: a multiply-defined external symbol (i.e., FORTRAN Common, or an uninitialized variable defined external to a function in C). In the symbol table of each file where the symbol is defined, the section number of the symbol will be zero and the value of the symbol will be a positive number giving the size of the symbol. When the files are combined, the link editor will combine all the input symbols into one symbol with the section number of the **.bss** section (or of the **.data** section, if one of the input symbols is initialized). The maximum size of all the input symbols with the same name will be used to allocate space for the symbol and the value will become the address of the symbol. This is the only case where a symbol may have a section number of zero and a non-zero value.

Symbols having certain storage classes are also restricted to certain section numbers. Table 5-21 lists these storage classes.

Table 5-21. Restricted Storage Classes	
Storage Class	**Section Number**
C_AUTO	N_ABS
C_EXT	N_ABS, N_UNDEF, N_SCNUM
C_STAT	N_SCNUM
C_REG	N_ABS
C_LABEL	N_UNDEF, N_SCNUM
C_MOS	N_ABS
C_ARG	N_ABS
C_STRTAG	N_DEBUG
C_MOU	N_ABS
C_UNTAG	N_DEBUG
C_TPDEF	N_DEBUG
C_ENTAG	N_DEBUG
C_MOE	N_ABS
C_REGPARM	N_ABS
C_FIELD	N_ABS
C_BLOCK	N_SCNUM
C_FCN	N_SCNUM
C_EOS	N_ABS
C_FILE	N_DEBUG
C_ALIAS	N_DEBUG

Type Field (n_type). The type field contains information about the basic and derived type for the symbol. Each symbol has one basic (or fundamental) type but can have more than one derived type. The format of the type entry is:

Bit:	15 14	13 12	11 10	9 8	7 6	5 4	3 0
Field:	d6	d5	d4	d3	d2	d1	type

The type field, bits 0—3, indicates one of the basic types listed in Table 5-22.

Table 5-22. Fundamental Types

Type	Mnemonic	Value
Type not assigned	T_NULL	0
Character	T_CHAR	2
Short integer	T_SHORT	3
Integer	T_INT	4
Long integer	T_LONG	5
Floating point	T_FLOAT	6
Double word	T_DOUBLE	7
Structure	T_STRUCT	8
Union	T_UNION	9
Enumeration	T_ENUM	10
Member of enumeration	T_MOE	11
Unsigned character	T_UCHAR	12
Unsigned short	T_USHORT	13
Unsigned integer	T_UINT	14
Unsigned long	T_ULONG	15

Bits 4—15 are arranged as six 2-bit fields marked d1 through d6. These d fields represent levels of the derived types listed in Table 5-23.

Table 5-23. Derived Types

Type	Mnemonic	Value
No derived type	DT_NON	0
Pointer	DT_PTR	1
Function	DT_FCN	2
Array	DT_ARY	3

The order of the derived types is from most tightly bound type to least tightly bound type (generally from right to left in the C language declaration).

The following examples demonstrate the interpretation of the symbol table entry representing type.

*char *func();*

Here *func* is the name of a function that returns a pointer to a character. The fundamental type of *func* is 2 (character), the d1 field is 2 (function), and the d2 field

is 1 (pointer). The type word in the symbol table for *func* would contain the hexadecimal number 0x62, which is interpreted to mean "function that returns a pointer to a character".

*short *tabptr[10][25][3];*

Here *tabptr* is a three-dimensional array of pointers to short integers. The fundamental type of *tabptr* is 3 (short integer); the d1, d2, and d3 fields each contains a 3 (array), and the d4 field is 1 (pointer). Therefore the type entry in the symbol table would contain the hexadecimal number 0x73f3, indicating a "three-dimensional array of pointers to short integers".

Table 5-24 shows which type entries are legal for each storage class.

Table 5-24. Storage Class Type Entries

Storage Class	Function?	d Entry Array?	Pointer?	typ Entry Basic Type
C_AUTO	No	Yes	Yes	Note 1
C_EXT	Yes	Yes	Yes	Note 1
C_STAT	Yes	Yes	Yes	Note 1
C_REG	No	No	Yes	Note 1
C_LABEL	No	No	No	T_NULL
C_MOS	No	Yes	Yes	Note 1
C_ARG	Yes	No	Yes	Note 1
C_STRTAG	No	No	No	T_STRUCT
C_MOU	No	Yes	Yes	Note 1
C_UNTAG	No	No	No	T_UNION
C_TPDEF	No	Yes	Yes	Note 1
C_ENTAG	No	No	No	T_ENUM
C_MOE	No	No	No	T_MOE
C_REGPARM	No	No	Yes	Note 1
C_FIELD	No	No	No	Note 2
C_BLOCK	No	No	No	T_NULL
C_FCN	No	No	No	T_NULL
C_EOS	No	No	No	T_NULL
C_FILE	No	No	No	T_NULL
C_ALIAS	No	No	No	Note 3

Notes:
1. Any except T_MOE.
2. T_ENUM,T_UCHAR,T_USHORT,T_UINT,T_ULONG.
3. T_STRUCT,T_UNION,T_ENUM.

Conditions for the d entries apply to d1 through d6, except that it is impossible to have two consecutive derived types of function.

Although function arguments can be declared as arrays, they will be changed to pointers by default. Therefore no function argument can have array as its derived type.

Structure for Symbol Table Entry. The following C language structure declaration is currently used for symbol table entries:

```
struct syment
{
        union
        {
                        char _n_name[SYMNMLEN];  /* symbol name */
                        struct
                        {
                                                 long _n_zeroes;        /* ==0 if in
                                                                           string table */
                                                 long _n_offset;        /* location in
                                                                           string table */
                        } _n_n;
                        char *_n_nptr[2];        /* allows overlaying */
        } -n:
        long            n_value;
        short           n_scnum;
        unsigned short  n_type;
        char            n_sclass;
        char            n_numaux;
};

#define  n_name         _n._n_name
#define  m_zeroes       _n._n_n._n_zeroes
#define  n_offset       _n._n_n._n_offset
#define  n_nptr         _n._n_nptr[1]
#define  SYMNMLEN       8
```

Auxiliary Table Entries

Currently, there is at most one auxiliary entry per symbol. The auxiliary table entry contains the same number of bytes as the symbol table entry. However, unlike symbol table entires, the format of the auxiliary table entry of a symbol depends on its type and storage class. These are summarized in Table 5-25.

Table 5-25 Auxiliary Symbol Table Entries				
Name	Storage Class	Type Entry		Auxiliary Entry Format
		d1	typ	
.file	C_FILE	DT_NON	T_NULL	File name
.text,.data,.bss	C_STAT	DT_NON	T_NULL	Section
tagname	C_STRTAG, C_UNTAG,C_ENTAG	DT_NON	T_NULL	Tag name
.eos	C_EOS	DT_NON	T_NULL	End of structure
fcname	C_EXT,C_STAT	DT_FCN	Note 1	Function
arrname	C_AUTO,C_STAT, C-MOS,C-MOU, C-TPDEF,C_EXT	DT_ARY	Note 1	Array
.bb,.eb	C_BLOCK	DT_NON	T_NULL	Beginning and end of block
.bf,.ef	C_FNC	DT_NON	T_NULL	Beginning and end of function
name related to structure, union, or enumeration	C_AUTO,C_STAT, C-MOS,C_MOU, C-TPDEF,C_EXT,	DT_PTR, DT_ARR, DT_NON	T_STRUCT T_UNION, T_ENUM	Name related to structure union, or enumeration

Note: Any except T_MOE.

In Table 5-25, the names tagname, fcname, and arrname represent any symbol name; however, only tagname can include the special symbol **.xfake**.

Any symbol that satisfies more than one condition in Table 5-25 should have a union format in its auxiliary entry. Symbols that do not satisfy any of the above conditions should NOT have any auxiliary entry.

File Names. Each of these auxiliary table entries contains a 14-character file name in bytes 0—13. The file names are padded with zeros to 14 characters.

Sections. The auxiliary table entries for sections have the format shown in Table 5-26. The remaining bytes are also filled with zeroes.

Table 5-26. Section Format		
Bytes	**Name**	**Contents**
0—3	x_scnlen	Section length
4—5	x_nreloc	Number of relocation entries
6—7	x_nlinno	Number of line numbers
8—17	—	0 (unused)

Tag Names. The auxiliary table entries for tag names have the format shown in Table 5-27.

Table 5-27. Tag Name Format

Bytes	Name	Contents
0—5	—	0 (unused)
6—7	x_size	Size of structure, union, or enumeration
8—11	—	0 (unused)
12—15	x_endndx	Index of next entry beyond this structure, union, or enumeration
16—17	—	0 (unused)

End of Structures. The auxiliary table entries for the end of structures have the format shown in Table 5-28.

Table 5-28. End of Structure Format

Bytes	Name	Contents
0—3	x_tagndx	Tag index
4—5	—	0 (unused)
6—7	size	Size of structure, union, or enumeration
8—17	—	0 (unused)

Functions. The auxiliary table entries for functions have the format shown in Table 5-29. If a function has been expanded in-line, it has the least significant bit of the x_fsize set to one in the auxiliary entry for its **.ef** symbol

Table 5-29. Function Format

Bytes	Name	Contents
0—3	x_tagndx	Tag index
4—7	x_fsize	Size of function
8—11	x_lnnoptr	File pointer to line number
12—15	x_endndx	Index of next entry beyond this function
16—17	—	0 (unused)

Arrays. The auxiliary table entries for arrays have the format shown in Table 5-30.

Table 5-30. Array Format		
Bytes	**Name**	**Contents**
0—3	x_tagndx	Tag index
4—5	x_lnno	Line number of declaration
6—7	x_size	Size of the array
8—9	x_dimen[0]	First dimension
10—11	x_dimen[1]	Second dimension
12—13	x_dimen[2]	Third dimension
14—15	x_dimen[3]	Fourth dimension
16—17	—	0 (unused)

End of Blocks and Functions. The auxiliary table entries for the end of blocks and functions have the format shown in Table 5-31.

Table 5-31. End of Block and Function Format		
Bytes	**Name**	**Contents**
0—3	—	0 (unused)
4—5	x_lnno	C source line number
6—17	—	0 (unused)

Beginning of Blocks and Functions. The auxiliary table entries for the beginning of blocks and functions have the format shown in Table 5-32.

Table 5-32. Beginning of Block and Function Format		
Bytes	**Name**	**Contents**
0—3	—	Unused
4—5	x_lnno	C source line number
6—11	—	Unused
12—15	x_endndx	Index of next entry past this block
16—17	—	0 (unused)

Names Related to Structures, Unions and Enumerations. The auxiliary table entries for structure, union, and enumeration names have the format shown in Table 5-33.

Table 5-33. Structure, Union, and Enumeration Format		
Bytes	**Name**	**Contents**
0—3	x_tagndx	Tag index
4—5	—	0 (unused)
6—7	x_size	Size of the structure, union, or enumeration
8—17	—	0 (unused)

Names defined by typedef may or may not have auxiliary table entries. For example:

```
typedef struct people STUDENT;
struct people  {
               char name[20];
               long id;
};
typedef struct people EMPLOYEE;
```

The symbol EMPLOYEE has an auxiliary table entry in the symbol table but the symbol STUDENT does not.

The C language data structure that is used to declare auxiliary symbol table entries can be found on the manual page for SYMS.H(5L) in **5.6 SGP MANUAL PAGES**.

5.4.8 String Table

Symbol table names longer than eight characters are stored contiguously in the string table with each symbol name delimited by a null byte. The first four bytes of the string table are the size of the string table in bytes; offsets into the string table will, therefore, be equal to or greater than four.

For example, given a file containing two symbols with names longer than eight characters, *long_name_1* and *another_one*, the string table would look like this:

28			
'l'	'o'	'n'	'g'
'_'	'n'	'a'	'm'
'e'	'_'	'1'	'\0'
'a'	'n'	'o'	't'
'h'	'e'	'r'	'_'
'o'	'n'	'e'	'\0'

5.5 UTILITIES AND LIBRARY ROUTINES

The output file obtained from the **m32as** assembler and the **m32ld** link editor is an object file named **m32a.out**. It has a format called the common object file format. The object file is executable if no errors or unresolved references are found. The file contains a header with size information, program sections, and a symbol table. Each section is composed of a section header, data, and relocation and line number information. Depending on the assembler or link editor options used to produce the object file, the file may be devoid of relocation entries, line number entries, the symbol table, or compiler-generated symbols.

The software generation programs (SGP) provide a variety of utilities to read and manipulate object files. Among the functions performed by the utilities are listing, reducing, or deleting various parts of an object file or symbol table. A library of interface functions that aid in the development of application programs is also provided as part of the SGP. Many projects will use the routines and data declarations that comprise the libraries. This library approach allows efficient, controlled development of common code and enhances portability.

The utilities are:

- **m32ar** Formats one or more files into a common archive file.
- **m32convert** Converts a *UNIX* System V archive file to *UNIX* System V archive file.
- **m32conv** Converts *WE* 32100 Microprocessor object files from one host machine format to another host machine format.
- **m32cprs** Compresses object files by removing duplicate structure and union descriptors from the symbol table.
- **m32lis** Produces assembly language listings from object files.
- **m32dump** Dumps selected parts of the named object files.
- **m32list** Produces a C language source listing with line numbers that specify where breakpoints can be inserted.
- **m32lorder** Generates an ordered listing of object files suitable for link editing in one pass, as done by **m32ld.**
- **m32size** Reports the number of bytes of text, uninitialized data, and initialized data (and their sum) included in an object module.
- **m32nm** Displays symbol table information.
- **m32strip** Reduces file storage overhead by removing symbolic testing information from an object file.

Because the SGP runs under the *UNIX* Operating System, the utilities can use the many features of shell commands. I/O redirection, pipes and filters, and the asynchronous capability of the shell are commonly used with the utilities. The defining of procedures and shell variables, the use of metacharacters, or other features of the shell may also prove useful.

The library of accessing routines provides an alternative method for examining parts of an object file. Specific applications may need to examine the contents of an object file from within a C language program. Although these programs must know the detailed structure of the parts of the object file that they process, the access routines insulate these calling programs from detailed knowledge of the overall structure of the object file. The accessing library is described at the end of this section.

5.5.1 Utility Programs

The manual pages for the utilities described here are found in **5.6 SGP MANUAL PAGES**. These manual pages were valid at the time of publication. The current manual pages can be obtained on-line from the *UNIX* System by using the **m32man** command. The manual page contains an explanation of each utility and lists the temporary files accessed.

One error message common to many utilities is "can't open file", meaning a file cannot be read. The message is usually caused by misspelling the file name or being in the wrong directory. A list of other commonly encountered error messages can be found with each utility description.

m32ar

The **m32ar** utility maintains groups of files combined into a single archive file. Its main use is to create and update library files as used by the link editor. When **m32ar** creates an archive, it creates headers in a format that is portable across all machines. The archive symbol table is used by the link editor to effect multiple passes over libraries of object files in an efficient manner. An archive symbol table is only created and maintained by **m32ar** when there is at least one object file in the archive. The archive symbol table is in a specially named file which is always the first file in the archive. This file is never mentioned or accessible to the user. Whenever **m32ar** is used to create or update the contents of such an archive, the symbol table is rebuilt. The *s* option, described in Table 5-34, forces the symbol table to be rebuilt.

To invoke **m32ar** the command line

m32ar key [posname] afile [name] \f3m32convert. . .

is used. Key is an optional –, followed by one character from the set *drqtpmx*, optionally concatenated with one or more of the set *vuaibcls*. Table 5-34 defines each character that can be used in the key argument. *Afile* is the archive file name. The *names* are constituent files in the archive file. The optional *posname* argument is described in Table 5-34 for the options that use it.

Table 5-34. m32ar Command Line Keys	
Key	**Description**
c	Suppresses the message that is produced by default when *afile* is created.
d	Deletes the named files from the archive file.
l	Places temporary files in the local current working directory, instead of the directory specified by the environment variable TMPDIR or in the default directory /tmp.
m	Moves the named files to the end of the archive file. If an optional positioning character from the set *abi* is used, then the *posname* argument must be present and specifies if files are moved after (*a*) or before (*b* or *i*) *posname*.
p	Prints the named files in the archive file.
q	Appends the named files to the end of the archive file. This option does not check whether the added members are already in the archive file.
r	Replaces the named files in the archive file. If the optional character *u* is used with *r*, then only those files with dates of modification later than the archive file are replaced. If a positioning character is present, then the *posname* argument must be present and, as in *m*, specifies where the files are to be positioned.
s	Forces the regeneration of the archive system table even if **m32ar** is not invoked with a command which will modify the archive contents. This command is useful to restore the archive symbol table after the **m32strip** command has been used on the archive.
t	Prints a table of contents of the archive file. If no names are given, all files in the archive are tabled. If names are given, only those files are tabled.
v	Gives a verbose file-by-file description of the making of a new archive file from the old archive and the constituent files. When used with *t*, gives a long listing of all information about the files. When used with *x*, precedes each file with a name.
x	Extracts the named files. If no names are given, all files in the archive are extracted. In neither case does *x* alter the archive file.

m32convert

The **m32convert** utility is used to transform an input file to an output file. An input file may be in any of the following four forms:

- A pre *UNIX* System Release 5.0 *VAX* Computer object file or link-edited (a.out) module (only with the *−5* option),
- A pre *UNIX* System Release 5.0 *VAX* Computer archive of object files or link edited (a.out) modules (only with the *−5* option),
- A pre *UNIX* System Release 5.0 3B20S Computer archive of object files or link edited (a.out) modules (only with the *−5* option), or
- A *UNIX* System Release 5.0 *VAX* Computer or 3B20S Computer archive file (without the *−5* option).

m32convert is used to transfer the input files into the following output files, respectively:

- an equivalent *UNIX* System Release 5.0 *VAX* Computer object file or link edited (a.out) module (with the *−5* option),
- an equivalent *UNIX* System Release 5.0 *VAX* Computer archive of equivalent object files or link edited (a.out) modules (with the *−5* option),
- an equivalent *UNIX* System Release 5.0 archive of unaltered 3B20S Computer object files or link edited (a.out) modules (with the *−5* option), and
- an equivalent *VAX* Computer or 3B20S Computer *UNIX* System Release 5.0 portable archive containing unaltered members (without the *−5* option).

All other types of input to the **m32convert** command will be passed unmodified from the input file to the output file (along with appropriate warning messages). When transforming archive files with the *−5* option, the **m32convert** command will inform the user that the archive symbol table has been deleted. To generate an archive symbol table, this archive file must be transformed again by **m32convert** without the *−5* option to create a *UNIX* System Release 5.0 archive file. Then the archive symbol table may be created by executing the **m32ar** command with the *ts* option. If a *UNIX* System Release 5.0 archive with an archive symbol table is being transformed, the archive symbol table will automatically be converted.

The command line used to invoke the **m32convert** command is

```
m32convert [ −5 ] infile outfile
```

where the option *−5* is used as described above. The argument *infile* is the input file and *outfile* is the output file. The name of the input and output files must be different.

m32conv

The **m32conv** utility is provided because the SGP runs on several machines. Whenever a file is moved from one machine to another with different architecture, **m32conv** should be used to format the resulting file.

Differences in byte ordering and data formats cause object file formats to differ in their symbolic information when produced on machines with different architectures. The **m32conv** utility converts a processor object file (e.g., **m32a.out**) from the internal format of one machine architecture to that with another architecture. For example, use **m32conv** to convert an object file produced on a 3B20S Computer to that for a *VAX* 11/780 Computer, or the resulting file will not be in a usable format. **m32conv** does not alter the contents of the **.text** or the **.data** sections; it only modifies the headers, symbol tables, and other symbolic information.

File conversion is necessary and effective between machines of the following three architectures:

1. A *DEC* Computer style byte ordering with 16-bit word length (e.g., *PDP* 11/70 Computer).
2. A *DEC* Computer style byte ordering with 32-bit word length (e.g., *VAX* 11/780 Computer).
3. An *IBM* Computer style byte ordering with 32-bit word length (e.g., *AT&T* 3B Computer).

The output of **m32conv** is a file having the same name as the input file with a suffix of **.v**. Output cannot be redirected from the **m32conv** command.

m32conv is best used within a procedure for sending object files from one machine to another. Attempting to convert a file when no conversion is necessary results in an error message, although the input file is copied to the output file. **m32conv** may be used on either the source (sending) or target (receiving) machine.

To use **m32conv**, enter the command line

m32conv [–] [–s] [–a| –o| –p] –t target *files*

where the **–t** option with a **target** name MUST be specified, and *files* is a list of files to be converted. Values recognized for the **–t** option are given in Table 5-35. The **b16** target indicates an 8086 microcomputer.

The **–a**, **–o**, and **–p** options indicate which archive format is to be used for the output file if the input file is an archive. The **–p** option produces an archive file in the *UNIX* System Release 5.0 random access archive format. This is the default. The **–a** option produces the output file in the *UNIX* System V Release 2.0 portable archive format. The **–o** option will produce archive in the old (pre-Release 5.0) archive format. **m32conv** will accept input archive files in all three formats.

Two other options, – and **–s**, are available. The minus sign by itself specifies that filenames are taken from the standard input. The **–s** option causes **m32conv** to function exactly as the *UNIX* System **swab** command, which exchanges adjacent odd- and even-numbered bytes. This may be useful depending on the actual transfer method used and the byte-ordering of the host machine.

Table 5-35. m32conv Target Machines	
pdp	*DEC PDP* 11/70 Minicomputer
vax	*DEC VAX* 11/780 Minicomputer
ibm	*IBM* 370 Computer
i80	8080 microcomputer
x86	8086 microcomputer
b16	8086 microcomputer with Basic-16
n3b	*AT&T* 3B Computer
m32	*WE* 32100 Microprocessor

All diagnostics are self-explanatory. Fatal errors on the command line cause the program to terminate. Fatal errors within an input file cause the program to continue at the next input file.

m32cprs

The **m32cprs** utility reduces the size of a processor object file by removing duplicate structure and union descriptors from its symbol table. To invoke this utility, enter the SGP command line

m32cprs *options file1 file2*

where *file1* is the input file, *file2* is the output file, and the available *options* are **−p** and **−v**. The input file is not changed by this process; the output file, where the compressed version of the input is placed, must be specified by the user.

The **−p** option causes the printing of statistical messages including: total number of tags, total duplicate tags, and total reduction of the size of *file1*. The **−v** option causes verbose error messages to be printed if an error occurs.

Some of the most commonly encountered error messages are:

- **usage: m32cprs [−v] [−p]** *infile outfile*
  Caused by failure to specify names for both input and output files.
- **Infile has incorrect magic number error condition: no compression**
  Occurs when *infile* is not in processor object file format.
- **no duplicate tags**
- **unable to open** *infile*
- **unable to create** *outfile*

m32dis

The **m32dis** disassembler utility produces an assembly language listing for each object file specified as input. The listing has a two-column format; assembly language statements are in the right column and the corresponding hexadecimal object code and machine address of the code are in the left column.

The disassembler produces a facsimile of the assembly language file that was assembled to produce a given object file. **m32dis** provides a convenient method of obtaining a processor assembly language listing of C language source programs and for assembly language programs written in assembler code.

To invoke the disassembler, enter the command line

m32dis *options files*

where *options* are chosen from Table 5-36 and *files* represents a list of object files. If no *options* are specified, all sections containing text are disassembled.

Three features of the **m32dis** listing are:

1. The disassembler prints line numbers for each C source line where a breakpoint can be set in square brackets, (e.g., [5] shows the fifth source line where execution can be halted for debugging). The line numbers appear in the first column, on the left hand side of the the instruction corresponding to the line where a breakpoint can be inserted.
2. The disassembler prints C function names followed by parentheses (e.g., printf( ) for the function printf). The function names appear in the first column, one line above the instruction that begins the function.
3. The disassembler prints computed addresses within a section when control is to be transferred to those addresses. They are printed within triangular brackets (e.g., <40> is computed address 40). These addresses appear in the operand field of control transfer instructions following a relative displacement. The computed address is the sum of the relative displacement and the address of the instruction currently being disassembled.

Note that items 1 and 2 occur only if the information exists in the object file (e.g., the code was compiled by **m32cc** with the **−g** option and the information was not removed by a utility or link editor option).

Table 5-36. m32dis Command Line Options		
Option	**Argument**	**Description**
−d	*section*	Disassembles the named section as data, and prints the offset of the data from the beginning of the section.
−da	*section*	Disassembles the named section as data, and prints the actual address of the data.
−F	*function*	Disassembles single named functions in each object file that is specified on the command line.
−l	*string*	Disassembles the library file specified by *string*. For example, one would issue the command line **m32dis** −l x −l z to disassemble the libraries **libx.a** and **libz.a**. The libraries are assumed to be in the SGP *lib* directory.
−o	None	Prints numbers in octal; without this option, default is hexadecimal.
−t	*section*	Disassembles the named section as text.
−V	None	Prints the version number of the disassembler being executed.

Note: Arguments are appended to options with no embedded blanks, except for the **−l** option.

The **−d** option causes the named section of the object file to be disassembled as a data section. The object code and its address relative to the beginning of the section are listed. **m32dis** makes no attempt to determine the corresponding assembly language statement. Addresses relative to the beginning of the named section are printed on the left side; object code bytes are printed on the right side, eight bytes per line.

The **−da** option causes disassembly of the named section of the object file as a data section. The object code and its absolute addresses are listed. No attempt is made to determine the corresponding assembly language statement.

If the **−F** option is used, only those named functions from each file will be disassembled.

The **−t** option causes the named section of the object file to be disassembled as a text section. The listing consists of the object code, its machine address, and the assembly language statements that produced the code. For example, if the command line is

m32dis −*t section files*

then the bytes of that section of object code are assumed to be opcode and operand encodings. The opcodes are looked up in the opcode disassembly table, and the operands are disassembled and printed.

The following is a list of error messages commonly encountered while executing the disassembler:

- **m32dis: <filename>: CANNOT OPEN:**
  Means the input file cannot be read.
- **m32dis: <filename>: BAD MAGIC NUMBER**
  The input file is not a processor object file.
- **m32dis: <filename> CANT FIND SECTION <section name>**
  An unknown section has been specified by the **−t** or **−d** option.
- **m32dis: <filename>: CANT FIND SECTION HEADER <section name>**
  The input file is not a processor object file or the file was not properly converted using **m32conv**.
- **m32dis: BAD FLAG <flag>**
  An unrecognized option has been specified.
- **m32dis: PREMATURE EOF**
- **m32dis: QUIRK--DATA SECTION HAS LINE NUMBER ENTRIES**

If the disassembler cannot find an opcode in the disassembler opcode lookup table, the message

```
ERROR UNKNOWN OPCODE
```

is printed on the same line as the bad object code and the disassembler then attempts to resynchronize itself. There are three cases determining how the disassembler resynchronizes.

1. If the file has been stripped of line number information as well as the symbol table, the following message is printed:

   ```
   NO LINE NUMBER ENTRIES EXIST
   NO SYMBOL TABLE EXISTS
   FOLLOWING DISASSEMBLY MAY BE OUT OF SYNC
   ```

   The disassembler will then continue with the two bytes immediately following the bad opcode.

2. If the file has been stripped of line number entries but has a symbol table, the following message is printed:

   ```
   NO LINE NUMBER INFORMATION EXISTS
   DUMP TO NEXT FUNCTION OR SECTION END
   IN ATTEMPT TO RESYNCHRONIZE
   ```

The disassembler then dumps bytes of object code until the next function or the section end (whichever comes first) is reached. At this point, the disassembler prints out:

DISASSEMBLER RESYNCHRONIZED

3. The file has line number entries. The disassembler then dumps bytes of object code until it reaches the next line where a breakpoint can be inserted. At that point the disassembler prints:

DISASSEMBLER RESYNCHRONIZED

m32dump

The **m32dump** utility allows examination of an object file by listing the contents of the file on standard output. The dump utility is normally used to look at different parts of an object file, with the parts being selected by options. **m32dump** attempts to format the information in a meaningful way by printing certain information in ASCII, hexadecimal, octal, or decimal as appropriate. The input file is unchanged after execution of **m32dump**, and no new files are created. **m32dump** accepts as input both object files and archive libraries of object files.

The options for **m32dump** are listed in Table 5-37. The **−a, −c, −f, −g, −h, −l, −o, −p, −r, −s, −t, −u,** and **−z** options specify which parts of an object file are to be dumped. These are the basic options, and can be used independently; others are modifying options. The options **−d, +d, −n, −t** (used with an argument), and **−z** (used with a numerical argument) are used in combination with other options to limit the range and type of information that is to be printed. The **−v** option is used to modify all but the **−o** and **−s** options. The **−v** option causes **m32dump** to interpret the information and print symbols instead of numbers; e.g., static instead of 0x03. The **−p** and **−o** options control the printing of header information.

Blanks separating an option and its modifier are optional. The comma separating the name from the number modifying the **−z** option may be replaced with a blank.

A simple example of **m32dump** is the command line

m32dump *−t m32a.out*

which would display the symbol table from the file **m32a.out**. The command line

m32dump *−tv m32a.out*

displays the symbol table from the file **m32a.out** in symbolic form. The command line

m32dump *−f −h −r −t 3 +t 10 test.o >testdump*

lists the file and section headers, the relocation information, and the symbol table entries three through ten for the object file test.o; the command also places the output in the file testdump.

Table 5-37. m32dump Command Line Options

Option	Argument	Description
−a	None	Dump the archive header of each member of each input archive file.
−c	None	Dump the string table.
−d	*number*	Dump the section number given or dump the range of sections beginning with the given number and ending either at the last section or at the number specified by **+d**.
+d	*number*	Dump only those sections having section numbers less than *number*. Begin either with the first section or with the section specified by the **−d** option.
−f	None	Dump each file header.
−g	None	Dump the global symbols in the symbol table of a *UNIX* System release 6.0 archive file.
−h	None	Dump all section headers.
−l	None	Dump line number information.
−n	*name*	Dump only the information pertaining to the named entity. This option is used with **−h, −s, −r, −l,** and **−t**.
−o	None	Dump each optional header.
−p	None	Suppress printing of the headers.
−r	None	Dump relocation information.
−s	None	Dump section contents.
−t	None	Dump symbol table entries.
−t	*index*	Dump only the indexed symbol table entry. **−t** used with the **+t** option specifies a range of symbol table entries.
+t	*index*	Dump symbol table entries in the range ending with the indexed entry. The range begins at the first symbol table entry or at the entry specified by the **−t** option.
−u	None	Underline the name of the file emphasis.
−v	None	Print symbolic, rather than numeric, information.
−z	*name*	Dump line number entries for the named function.
−z	*name,number*	Dump line number entry or range of line numbers starting at number for the named function.
+z	*number*	Dump line numbers starting at either the function name or number specified by **−z** up to number specified by **+z**.

The more common error messages produced by **m32dump** are:

- **usage: m32dump [flags] file ...**
  Occurs when the object file to be dumped is not named.
- **m32object: bad magic *file.out***
  Occurs when the file *file.out* is not a *WE* 32100 Microprocessor object file.
- **m32object: cannot open *file.out***
  Means *file.out* cannot be read.
- **m32dump: unknown option** *OPTION*.

m32list

The **m32list** utility lists C source files with line number information attached. **m32list** uses the object file corresponding to the input C language source file to determine the lines where breakpoints can be set. Generally breakpoints can be set at each executable statement that begins a new line of source code.

To invoke the processor list utility, use the command

m32list [−V][−h] *source [source...][object]*

where the square brackets denote optional entries, *source* is the source file name, and *object* is the object file name. If several C source files were used to create the object file, then a list of source files should be input to **m32list**. The last name in the list of files is considered to be the name of the object file. The default object file, **m32a.out**, is used when no object file appears on the command line. The input object file MUST have symbolic debugging symbols for **m32list** to work.

Line numbers are printed for each compiler-generated line where a breakpoint can be inserted. Line numbering begins anew for each C language function. Line number 1 always indicates the line containing the left curly brace ({) that begins a function body. Line numbers are also printed for inner block redeclarations of local variables so that those variables can be distinguished by the symbolic debugger.

The **−h** option suppresses the printing of headings.

The −V option prints the version of **m32list** being executed.

Object files that have no line numbers cause an error message to be printed. Because **m32list** does not use the C preprocessor, it may not recognize function definitions whose syntax has been distorted by the use of C preprocessor macro substitutions.

Some errors commonly encountered when using **m32list** are:

- **usage: m32list sourcefile [sourcefile...][object file]**
  Caused when no object file or no source file is specified.
- **m32list: name: cannot open**
  Caused when an input file name cannot be read.

- **m32list: unknown option *option***

m32lorder

The **m32lorder** library orders object file libraries for the link editor, **m32ld**. If the archive members are arranged by **m32lorder** so that every symbol and function is defined after it is referenced, **m32ld** will make fewer passes over the library and will therefore be more efficient. The SGP command line for library ordering works the same way as its *UNIX* System counterpart. To invoke the library ordering utility, use the command line

m32lorder *files*

where *files* indicates the input of one or more object or library archive files. The **m32lorder** output is a list of pairs of object file names, where the first file of the pair contains references to external identifiers defined in the second. Therefore, the second member of the pair must appear after the first to be properly loaded.

The names of input object files *must* end with **.o**, even when contained in library archives. Files with names not adhering to this rule have their global symbols and references attributed to some other file, and nonsense results.

The **m32lorder** output may be processed by the *UNIX* System **tsort** command to find an ordering of a library suitable for one-pass access by the **m32ld** link editor. The following example shows the use of **tsort**, along with **m32ar**, to build a new library from all existing files with names ending in **.o**. The archive library is named *libx.a* both before and after the operation:

m32ar cr libx.a 'm32lorder *.o | tsort'

m32nm

The **m32nm** name list utility displays the symbol table for each processor object file that is given as input. The input may be a relocatable or an absolute processor object file; or it may be an archive library of relocatable or absolute object files.

For each symbol in the table, the following information is printed:

Name the name of the symbol.

Value the symbol value expressed as an offset or an address depending on storage class.

Class the symbol storage class.

Type the symbol type and derived type. If the symbol is an instance of a structure or of a union, then the structure or union tag is given following the type (e.g., struct-person where person is the structure tag). If the symbol is an array, then the array dimensions are given following the type (e.g., char[n][m]).

Size the symbol size in bytes, if applicable. Special symbols have undefined size.

Line the source line number where it is defined, if applicable.

Section for storage classes static and external, the object file section containing the symbol.

m32nm does not change the input file and produces no new file. The syntax to invoke the name list utility is

m32nm *options filenames*

where *options* are chosen from Table 5-38 and *filenames* are the names of the input file(s) and/or archive(s).

Table 5-38. m32nm Command Line Options

Option	Description
−e	Prints only static and external symbols.
−f	Produces full output. Redundant symbols (.text, .data, and .bss) normally suppressed, are printed.
−n	Sorts the external symbols by name before printing them.
−o	Prints the value and size for each symbol in octal instead of the normal decimal.
−T	Truncates very long names.
−u	Prints only the undefined symbols.
−v	Sorts external symbols by value before printing them.
−V	Prints the version name of **m32nm** that is executing.
−x	Prints the value and size in hexadecimal.

The options may be specified in any order, either singly or in combination, and may appear anywhere on the command line. Therefore, both **m32nm** *name −e −v* and **m32nm** *−ve name* print the static and external symbols in *name*, with the external symbols sorted by value. If neither the **−n** nor the **−v** option is specified, the external symbols are printed in the order in which they are encountered.

Some common error messages that **m32nm** produces are:

- **usage: m32nm:** ***file:*** **bad magic**
  Input file is not a *WE* 32100 Microprocessor object file.
- **m32nm: name: cannot open**
  Input file cannot be read.
- **m32nm: name: bad magic**
  Input file is not a processor object file.
- **m32nm: name: no symbols**
  Symbols were stripped from the input file before it was input to **m32nm.**
- **m32nm: unknown option** ***option***

m32size

m32size prints the number of bytes required for each section (e.g., **.text,.data,** and **.bss**) of the input processor object file and the total number of bytes for all three sections. Such information may be needed for locating sections in memory.

The file input to **m32size** remains unchanged. The output consists of the name of each section, followed by its size in bytes, its physical address, and its virtual address. The form of the command line for **m32size** is:

m32size [−o][−d][−V] *filename[s]*

By default, numbers are printed in hexadecimal. The **−d** option specifies decimal numbers; the **−o** option specifies octal. Version information is printed when the **−V** option is specified.

Commonly encountered diagnostics are:

- **m32size: filename: cannot open**
  Occurs when *filename* cannot be read.
- **m32size: filename: bad magic**
  Occurs when *filename* is not a *WE* 32100 Microprocessor object file.

m32strip

The **m32strip** strip utility removes the symbol table and line number information from processor object files and archive libraries, thus saving space. The effect of **m32strip** is the same as the **−s** option of **m32ld**. After a file has been stripped, no symbolic debugging access is available for that file. This command should be run only on production versions of object files that have been debugged and tested.

The command line used to strip symbol table and line numbers is

m32strip [−l][−x][−r][−V] *name...*

where *name* is the name of a processor object file or archive library. Any number of *names* may be specified. If *name* is an archive, **m32strip** removes the local symbols from each object module in the archive. By deleting these symbols, the size of the archive is decreased and link-editing performance improves.

The amount of information stripped from the symbol table can be controlled by using either the **−l** or the **−x** options. With the **−l** option, only line number information is stripped. Symbol table information remains unchanged. With the **−x** option, no static or external symbol information gets stripped. The **−V** option prints version information.

If there are any relocation entries in the object file and symbol table information is to be stripped, **m32strip** terminates without stripping *name* and prints the error message:

m32strip: name: relocation entries present; cannot strip

The **−r** option allows the user to override this warning and force **m32strip** to strip an object module even if the module contains relocation information. When the **−r** option is used, **m32strip** will strip only local symbols and line number information. It will retain the global and static symbols and relocation information needed for link editing.

Other commonly encountered error messages are:

- **m32strip: name: cannot open**
  Occurs when *name* cannot be read.
- **m32strip: name: bad magic**
  Occurs when *name* is not a *WE* 32100 Microprocessor object file.
- **usage: m32strip [−l][−x][−r] file...**
  Occurs when no input file was specified.

5.5.2 Accessing Library

A library of object file access routines is available to aid in the development of application programs. Specific applications may need to examine the contents of an object file from within a C language program. Although these programs must know the detailed structure of the parts of the object file that they process, the access routines insulate these calling programs from detailed knowledge of the overall structure of the object file.

The interface between the calling program and the object file access routines is based on the defined type **LDFILE**, defined as *struct ldfile*, and declared in the header file **ldfcn.h.** The primary purpose of this structure is to provide uniform access to both simple object files and to object files that are members of an archive library.

All library functions except *ldopen, ldaopen, ldtbindex*, *ldgetname*, *sgetl*, and *sputl* return either the constant SUCCESS, defined as 1, or FAILURE, defined as 0. *ldopen* and *ldaopen* both return pointers to a **LDFILE** structure, while *ldtbindex* returns an index to a symbol table entry. *ldgetname* returns a character pointer, *sgetl* returns a long integer, and *sputl* does not return a value.

Use of the Accessing Library

To use the object file access functions, a C language program must include at least the files *<stdio.h>*, "*INCDIR/filehdr.h*", and "*INCDIR/ldfcn.h*", as described on the manual pages for each function. If the path names present a problem, consult the manual page for **INTRO**. Any program that uses the object file access routines must also be loaded with the access routine library, **libld.a**. This is done by adding **-lld** on the final link edit line when compiling a program.

The functions comprising the accessing library can be accessed from assembly language code by simulating the C calling sequence. This is best accomplished by using the interface macros described under **Macro Processing Facilities**, found in **5.2.1 Assembler**.

An example, the assembly language function *getindex*, is defined here. This function calls the **ldtbindex** library routine and places the result (a symbol table index) in r0.

```
C_PROLOGUE(getindex)
                    C_CALL(ldtbindex,_ISTARG)
                    movw            %r0,%_RESULT
C_RETURN
```

Note that the *movw* statement is unnecessary as long as **_RESULT** is defined as register zero (this is currently true). Nevertheless, it is good practice to insulate code from potential changes through this type of statement.

Library Functions and Macros

The object file access functions may be divided into four categories:

1. Functions that open or close an object file
2. Functions that read header or symbol table information
3. Functions that seek to the start of the section, relocation, or line number information for a particular section
4. A function to return the index of a particular symbol table entry, *ldtbindex*.

Additional access to an object file is provided through a set of macros contained in the library. The operation of these macros parallels the standard input/output file reading and manipulating functions.

Functions That Open or Close Object Files. The functions *ldopen* and *ldaopen* open object files and archives of object files. These two functions, along with their counterparts for closing functions (*ldclose* and *ldaclose*), are designed to provide uniform access to both simple object files and object files that are members of archive files. Thus an archive of processor object files can be processed as if it were a series of simple processor object files.

The function *ldopen* allocates and initializes the **LDFILE** structure and returns a pointer to the **LDFILE** structure to the calling program. The fields of the **LDFILE** structure may be accessed individually through macros defined in the header file **ldfcn.h**, and contain the following information:

TYPE(ldptr)	the file magic number, used to distinguish between archive members and simple object files.
IOPTR(ldptr)	the file pointer returned by the *UNIX* System function **fopen** and used by the standard input/output functions.
OFFSET(ldptr)	the file address of the beginning of the object file; the offset is nonzero if the object file is a member of an archive file.
HEADER(ldptr)	the file header structure of an object file.

In addition to the **#include** files, the functions that open or close files must be declared within the user program. For example,

```
LDFILE    *ldopen( ), *ldaopen( );
```

ldopen and *ldaopen* both take two arguments, *filename*, a pointer to a character string, and *ldptr*, a pointer to an **LDFILE** structure. If *ldptr* has the value **NULL**, *ldopen* opens the file *filename*, allocates and initializes the **LDFILE** structure, and returns a pointer (to the structure) to the calling program. If *ldptr* is valid and if **TYPE**(*ldptr*) is the archive magic number, *ldopen* reinitializes the **LDFILE** structure for the next member of the archive file, *filename*.

ldopen and *ldclose* are designed to work together. *ldclose* returns **FAILURE** only when **TYPE**(*ldptr*) is the archive magic number and there is another file in the archive to be processed. Only then should *ldopen* be called with the current value of *ldptr*. In all other cases, (particularly when a new file, *filename*, is opened), *ldopen* should be called with a **NULL** *ldptr* argument.

The following is a prototype for the use of *ldopen* and *ldclose*:

```
/* for each filename to be processed */

ldptr = NULL;
do
{
        if ((ldptr = ldopen(filename, ldptr)) != NULL)
        {
                /*check magic number */
                /*process the file */
        }
} while (ldclose(ldptr) == FAILURE);
```

If the value of *oldptr* is not **NULL,** *ldaopen* opens *filename* anew and allocates and initializes a new **LDFILE** structure, copying the **TYPE, OFFSET**, and **HEADER** fields from *oldptr*. *ldaopen* returns a pointer to the new **LDFILE** structure. This new pointer is independent of the old pointer, *oldptr*. The two pointers may be used concurrently to read separate parts of the object file. For example, one pointer may be used to step sequentially through the relocation information, while the other is used to read indexed symbol table entries.

Both *ldopen* and *ldaopen* open *filename* for reading. Both functions return **NULL** if *filename* cannot be opened, or if memory for the **LDFILE** structure cannot be allocated. Successfully opening a file does not insure that the given file is a processor object file or an archived object file.

If *TYPE(ldptr)* does not represent an archive file, *ldclose* closes the file and frees the memory allocated to the *LDFILE* structure associated with *ldptr*. If *TYPE(ldptr)* is the magic number of an archive file, and if there are any more files in the archive, *ldclose* reinitializes *OFFSET(ldptr)* to the file address of the next archive member and returns *FAILURE*. The *LDFILE* structure is prepared for a subsequent *ldopen*. In all other cases, ldclose returns *SUCCESS*.

ldaclose closes the file and frees the memory allocated to the *LDFILE* structure associated with *ldptr* regardless of the value of *TYPE(ldptr)*. *ldaclose* always returns *SUCCESS*. The function is often used in conjunction with *ldaopen*.

Functions That Read. Six functions read header or symbol table information. Five return either SUCCESS or FAILURE, and all must be loaded with the object file access library. Manual pages for each function are in **5.5 UTILITIES AND LIBRARY ROUTINES**. These functions are:

ldahread reads the archive header of a member of an archive file;

ldfhread reads the file header of a processor object file;

ldshread
ldnshread read an indexed or named section header of a processor object file, respectively;

ldtbread reads a symbol table entry of a processor object file;

ldgetname retrieves a symbol name from a symbol table entry or from the string table.

Functions That Seek. Eight functions position an object file at (i.e., seek to) the start of the section, or the relocation or line number information for a particular section. These functions point to, and thus identify, the parts of object files. All eight return either SUCCESS or FAILURE, and must be loaded with the object file access library, as previously described. Some unusual FAILUREs can occur when using these functions; consult the manual pages for details. The seeking functions are:

ldohseek points to the optional file header of an object file;

ldsseek
ldnsseek point to an indexed or named section of an object file, respectively;

ldrseek
ldrnseek point to the indexed or named relocation entries of a section of an object file;

ldlseek
ldlnseek point to the indexed or named line number entries of a section of an object file;

ldtbseek points to the symbol table of an object file.

Function That Returns the Index of a Symbol Table Entry. The function *ldtbindex* returns the index of a symbol table entry. This index may be used in subsequent calls to *ldtbread*. However, because *ldtbindex* returns the index of the symbol table entry that begins at the current position of the object file, calling *ldtbindex* after a particular symbol table entry has been read causes the index of the next entry to be returned.

The function *ldtbindex* fails if there are no symbols in the object file or if the object file is not positioned at the beginning of a symbol table entry. Note that the first symbol in the symbol table has an index of zero. Consult the manual page for additional usage details.

Macros. A set of macros defined in *ldfcn.h* provides additional access to object files. The macros parallel the standard input/output file reading and manipulating functions, translating a reference in the **LDFILE** structure into a reference to its file descriptor field. The following macros are provided:

GETC	get a character from a stream (same as C language's **getchar**);
FGETC	a function to retrieve a character from a stream;
GETW	get a word from a stream;
UNGETC	push a character back onto the input stream;
FGETS	get a string from a stream;
FREAD	buffered binary input/output;
FSEEK	set the position of the next input or output operation on a stream;
FTELL	obtain an offset for **FSEEK**;
REWIND	reposition a stream;
FEOF	tell when end of file is read on an input stream;
FERROR	tell when an error has occurred in reading or writing a stream;
FILENO	return the file descriptor associated with a stream;
SETBUF	assign buffering to a stream.
STROFFSET	calculates the address of the string table in an object file.

These macros and functions are described on the manual page **LDFCN** and are essentially the same as the standard *UNIX* System input/output library functions.

Note: The macro **FSEEK** translates into a call to the standard *UNIX* System input/output function, **fseek**(3S). The macro **FSEEK** should not be used to seek to the end of an archive file, because the end of an archive file may not be the same as the end of one of its object file members.

5.5.3 General-Purpose Library

A general-purpose library is available with the SGP to provide the functions of I/O formatting and conversion, string operations, memory operations, searching, random number generation, absolute value calculation, encryption, and byte order conversion.

Use of the General-Purpose Library

To use routines from the general-purpose library, a C language program must include the header files described in the manual pages for the routines used. For instance, the memory access routines require the header <memory.h>. Any program that uses the general-purpose routines must be loaded with the general-purpose library, *libcm32.a*. This is done when compling the program by adding –lcm32 on the final link edit line.

The general-purpose routines can be accessed from assembly language code by simulating the C calling sequence. This is best accomplished by using the interface macros described in **Macro Processing Facilities** (found in **5.2.1 Assembler**), and as illustrated in **5.5.2 Accessing Library**.

Routines in the General-Purpose Library

The following routines comprise the general-purpose library:

Routine	Summary
a64l	convert base-64 ASCII strings to long integers
abs	return integer absolute value
atoi	convert string to integer
atol	convert string to long integer
bsearch	binary search a sorted table
crypt	generate DES encryption
ctype	table of character types
l3tol	convert 3-byte integers to long integers
l64a	convert long integers to base-64 ASCII strings
lfind	linear search and update routine
lsearch	line search routine
itoll3	convert long integers to 3-byte integers
memccpy	memory copy till character
memchr	return pointer to first occurrence of character
memcmp	compares first n characters of arguments
memcpy	copies n characters from memory
memset	sets first n characters to c
printf	print formatted output
rand	simple random-number generator
scanf	reads formatted input
sprintf	generates formatted strings
srand	initial random-number generator
sscanf	parses formatted strings
strcat	appends string
strchr	returns pointer to first occurrence of character c
strcmp	compares two strings lexigraphically
strcpy	copies string
strcspn	returns length of initial string segment not from string2
strlen	returns number of characters in string
strncat	appends at most n characters
strncmp	compares at most n characters
strncpy	copies at most n characters
strpbrk	returns pointer to first occurrence of character from string2
strrchr	returns pointer to last occurrence of character c
strspn	returns length of initial string segment from string2
strtok	returns pointer to next token
strtol	convert string to long integer
swab	swap bytes

toascii	zero out non-ASCII bits
tolower	translate to lower case
toupper	translate to upper case

Routines Required When Using printf and scanf

There are two routines which reference other functions that are not in the general purpose library: *printf* calls *putchar*, and *scanf* calls *getchar*.

If the *WE* 321EB Microprocessor Evaluation Board is the target on which the user's programs will be run, then the *putchar* and *getchar* routines are provided in the *WE* 321SE Evaluation Software Programs. If the user's target is the *WE* 321AP Microprocessor Analysis Pod, *putchar* and *getchar* are in the *WE* 321SD Development Software Programs.

Otherwise, to use *printf*, the user must supply *putchar* and to use *scanf*, the user must supply *getchar*. *putchar* accepts a characer and returns an int which is EOF (−1) on error. *getchar* returns an int that is EOF (−1) on end-of-file or error.

sprintf and *sscanf* do not require any additional routines.

5.6 SGP MANUAL PAGES

The manual pages for the command, subroutines and file formats that comprise the SGP are contained in this section. They were current at the time of publication and are similar to those obtained with the **man** command. Use the **man** command to obtain the manual pages that apply to your version of the SGP. Table 5-39 lists the manual pages that are in this section.

Table 5-39. SGP Manual Pages

Commands	Subroutines	File Formats
M32AR	A64L	INTRO
M32AS	ABS	FILEHDR
M32CC	BSEARCH	LDFCN
M32CONV	CONV	LINENUM.H
M32CONVERT	CRYPT	M32A.OUT
M32CPRS	CYTYPE	PATHS
M32DIS	L3TOL	RELOC
M32DUMP	LDAHREAD	SCNHDR
M32LD	LDCLOSE	SYMS
M32LIST	LDFHREAD	
M32LORDER	LDGETNAME	
M32MAN	LDLREAD	
M32NM	LSEARCH	
M32SIZE	LDLSEEK	
M32STRIP	LDOHSEEK	
	LDOPEN	
	LDRSEEK	
	LDSHREAD	
	LDSSEEK	
	LDTBINDEX	
	LDTBREAD	
	LDTBSEEK	
	MEMORY	
	PRINTF	
	RAND	
	SCANF	
	SPUTL	
	STRING	
	STROTL	
	SWAB	

Synopsis entries for the command manual pages list the command line. For the subroutines (libraries) the synopsis lists the information of the library file. The file formats synopsis lists the file(s) for the file format.

NAME

m32ar — archive and library maintainer for portable archives

SYNOPSIS

m32ar key [posname] afile [name] ...

DESCRIPTION

The *m32ar* command maintains groups of files combined into a single archive file. Its main use is to create and update library files as used by the link editor. It can be used, though, for any similar purpose. The magic string and the file headers used by *m32ar* consist of printable ASCII characters. If an archive is composed of printable files, the entire archive is printable.

When *m32ar* creates an archive, it creates headers in a format that is portable across all machines. The portable archive format and structure is described in detail in *m32ar*. The archive symbol table is used by the link editor (*m32ld*) to effect multiple passes over libraries of object files in an efficient manner. An archive symbol table is only created and maintained by *m32ar* when there is at least one object file in the archive. The archive symbol table is in a specially-named file which is always the first file in the archive. This file is never mentioned or accessible to the user. Whenever the *m32ar* command is used to create or update the contents of such an archive, the symbol table is rebuilt. The *s* option described below will force the symbol table to be rebuilt.

Key is an optional **−m**, followed by one character from the set *drqtpmx*, optionally concatenated with one or more of *vuaibcls*. *Afile* is the archive file. The *names* are constituent files in the archive file. The meanings of the key characters are:

d Delete the named files from the archive file.

r Replace the named files in the archive file. If the optional character *u* is used with *r*, then only those files with dates of modification later than the archive files are replaced. If an optional positioning character from the set *abi* is used, then the *posname* argument must be present and specifies that new files are to be placed after (*a*) or before (*b* or *i*) *posname*. Otherwise new files are placed at the end.

q Quickly append the named files to the end of the archive file. Optional positioning characters are invalid. The command does not check whether the added members are already in the archive. Useful only to avoid quadratic behavior when creating a large archive piece-by-piece.

t Print a table of contents of the archive file. If no names are given, all files in the archive are tabled. If names are given, only those files are tabled.

p Print the named files in the archive.

m Move the named files to the end of the archive. If a positioning character is present, then the *posname* argument must be present and, as in *r*, specifies where the files are to be moved.

x Extract the named files. If no names are given, all files in the archive are extracted. In neither case does *x* alter the archive file.

v Give a verbose file-by-file description of the making of a new archive file from the old archive and the constituent files. When used with *t*, give a long listing of all information about the files. When used with *x*, precede each file with a name.

c Suppress the message that is produced by default when *file* is created.

l Place temporary files in the local current working directory, rather than in the directory specified by the environment variable *IMPDIR* or in the default directory *\tmp*.

s Force the regeneration of the archive symbol table even if *m32ar* is not invoked with a command which will modify the archive contents. This command is useful to restore the archive symbol table after the *m32strip* command has been used on the archive.

FILES

\tmp\ar* temporaries

SEE ALSO

m32convert, m32ld, m32lorder, m32strip, m32a.out

NOTES

This archive format is new to this release. The *m32convert* command can be used to change an older archive file into an archive file that is recognized by this *m32ar* command.

BUGS

If the same file is mentioned twice in an argument list, it may be put in the archive twice.

NAME

m32as — *WE* 32100 Microprocessor Assembler

SYNOPSIS

m32as [**−o** objfile] [**−n**] [**−m**] [**−R**] [**−V**] file−name

DESCRIPTION

The **m32as** command assembles the named file.

The following flags are recognized by the assembler and may be specified in any order:

−o *objfile*
Output of assembly is put in *objfile*. By default, the output file name is formed by removing the **.s** suffix, if there is one, from the input file name and appending a **.o** suffix.

−n Turns off long/short address optimization. By default, address optimization takes place.

−m Invokes the *m4* macro processor. By default, does not invoke *m4* on the input to the assembler.

−R Remove (unlink) the input file after assembly is completed.

−V Causes the version number of the assembler being run to be written on standard error.

FILES

/usr/tmp/m32as[1-6]XXXXXX temporary files

SEE ALSO

m32ld,**m32nm**,**m32strip**, **m32a.out**.

DIAGNOSTICS

If the input file cannot be read, the assembly terminates with the message "Unable to open input file". If assembly errors are detected in the input file the following information is written to standard error: the input file name, line number where the error occurred in the assembly code, a descriptive message of the problem, and, if the input file was produced by the C compiler (see *m32cc*), and the line number in the C program that generated the erroneous code.

CAVEATS

If the input file does not contain a **.file** assembler, then the file name given by the assembler when an error occurs is one of the temporary files.

If the *m4* macro processor (see **5.2.1 Assembler**) is used, then *m4* keywords cannot be used as symbols (variables, functions, labels) in the input assembly file, since *m4* cannot determine which are assembler symbols and which are real *m4* macros.

BUGS

The **.align** assembler directive is not guaranteed to work in the **.text** section when optimization is performed. Arithmetic expressions may have only one forward-referenced symbol per expression.

NAME

m32cc — *WE* 32100 Microprocessor C Compiler

SYNOPSIS

m32cc [−c] [−p] [−g] [−y] [−O] [−S] [−P] [−E] [−V] [−Dsymbol**]...[−U**symbol**]**
...[I*dir*] files

DESCRIPTION

The **m32cc** command is the interface to the C compiler, assembler, and link editor. Arguments whose names end with **.c** are taken to be C source programs and those with **.s** are taken as assembly programs; they are compiled/assembled, and link edited. The resulting object and code is left in a file named m32a.out.

The following flags are interpreted by **m32cc**. See **m32ld** or **m32as** for other useful flags.

−c	Run the preprocessor, compiler, and assembler, and leave the object code on corresponding files suffixed with **.o**.
−p	This flag is reserved for invoking a profiler.
−g	Produce additional information needed for the use of **sdb**.
−y *limit*	Set limit on percent growth per file due to in-line expansion. Values for *limit* are: *u*, allows unlimited growth; integer >0 allows indicated percent growth; *s*, suppresses in-line expansion.
−O	Invoke an object-code optimizer. The optimizer will move, modify, merge and delete code, so symbolic debugging with line numbers could be confusing when the optimizer is used.
−S	Compile the named C programs, and leave the assembler-language output on corresponding files suffixed **.s**.
−P	Run only the macro preprocessor on the named C programs, and leave the output on corresponding files suffixed **.i**.
−E	Same as the **−P** option except the output is directed to the standard output. This allows the preprocessor to be used as a filter for any other compiler.
−V	Print the version of the compiler, optimizer, assembler or link-editor that is invoked.
−D	Define *symbol* to the preprocessor. This mechanism is useful with the conditional statements in the preprocessor by allowing symbols to be defined external to the source file.

−U	Undefine *symbol* to the preprocessor.
−I	Change the algorithm for searching for **#include** files whose names do not begin with / to look in *dir* before looking into the directions on the standard list. Thus, **#include** files whose names are enclosed in " " will be searched for first in the directory of the file argument, then in directories named in **−I** options, and last in directories on a standard list. For **#include** files whose names are enclosed in <>, the directory of the *file* argument is not searched.
−Wc, arg1 [,arg2 ...]	Hand off the argument[s] *m32argn* to pass **c** where **c** is one of [p02a1] indicating preprocessor, compiler, optimizer, assembler, and link editor, respectively. For example, **−Wa**, **−m** invokes the m4 macro preprocessor on the input to the assembler.
−B string	Construct pathnames for substitute preprocessor, compiler, assembler, and link editor passes by concatenating string with the suffixes cpp, comp, optim, m32as, m32ld.
−t [p02a1]	Find only the designated preprocessor, compiler, assembler, and link edit passes in the file whose names are constructed by a **−B** option. " " is equivalent to *−tp02*.

Other arguments are taken to be either link-editor flag arguments, or C compatible object programs, typically produced by an earlier **m32cc** run, or perhaps libraries of C compatible routines. These programs, together with the results of any compilations specified, are link-edited (in the order given) to produce an executable program with name **m32a.out** unless the **−o** option of the link-editor is used.

FILES

File	Description
file.c	input file
file.o	object file
file.s	assembly language file
m32a.out	link-edited output
/usr/tmp/m32?	temporary
LIBDIR/comp	compiler
LIBDIR/optim	optimizer
LIBDIR/libc.a	*WE* 32100 Microprocessor Library

SEE ALSO

m32as, m32dis, m32ld, m32list.

DIAGNOSTICS

The diagnostics produced by the C compiler are sometimes cryptic. Occasional messages may be produced by the assembler or link-editor.

NOTES

By default, the return value from a C program is completely random. The only two guaranteed ways to return a specific value is to explicitly call exit(2) or to leave the function main() with a "return expression;" construct.

NAME

m32conv — *WE* 32100 Microprocessor SGP Object File Converter

SYNOPSIS

m32conv [−][−s] [−a|−0|−p] −t target files

DESCRIPTION

The **m32conv** command converts object files from their current format to the format of the *target* machine. The converted file is written to file.v.

Command line options are:

−	indicates *files* should be read from *stdin.*
−a	If the input file is an archive, produce the output file in the *UNIX* System V Release 2 portable archive format.
−o	If the input file is an archive, produce the output file in the old (pre *UNIX* System Release 5.0) archive format.
−p	If the input file is an archive, produce the output file in the *UNIX* System V Release random access archive format. This is the default.
−s	causes **m32conv** to function exactly as the *UNIX* System **swab** command. This is useful only for 3B20 object files which are to be "swab-dumped" from a *DEC* Computer to a 3B20 Computer.
−t target	indicates the machine (target) to which the object file is being shipped. This may be another host or a target machine. Legal values for target are: pdp, vax, ibm, i80, x86, b16, n3b and m32.

m32conv can be used to convert all object files in common object file format, not only object files. it can be used on either the source (sending) or target (receiving) machine.

m32conv is meant to ease the problems created by a multihost cross-compilation development environment. **m32conv** is best used within a procedure for shipping object files from one machine to another.

m32conv will recognize and produce archive files in three formats: the pre *UNIX* System Release 5.0 format, the 5.0 random access format, and the System V Release 2 portable ASCII format.

EXAMPLE

```
*ship object files from vax to ibm
$echo *.out|m32conv −t ibm −$OFC/foo.o
$uucp *.v my370!~/rje/
```

DIAGNOSTICS

All intended to be self-explanatory. Fatal diagnostics on the command lines cause termination. Fatal diagnostics on an input file cause the program to continue to the next input file.

BUGS

Special applications must compile **m32conv** differently if it is to convert special object files, e.g., products of **ldp**, correctly. **m32conv** will not convert archives from one format to another if both the source and target machines have the same byte ordering. The *UNIX* System tool *m32convert* should be used for this purpose.

NAME

m32convert — convert object and archive files to common formats

SYNOPSIS

m32convert [−5] infile outfile

DESCRIPTION

m32convert transforms input *infile* to output *outfile*. *Infile* must be different from *outfile*. The *−5* option causes *m32convert* to work exactly as it did for *UNIX* System Release 5.0. *Infile* may be any one of the following:

1. a pre *UNIX* System Release *VAX* Computer object file or link-edited (a.out) module (only with the *−5* option).
2. a pre *UNIX* System Release *VAX* Computer archive of object files or link-edited (a.out) modules (only with the *−5* option).
3. a pre *UNIX* System Release 3B20S Computer archive of object files or link-edited (a.out) modules (only with the *−5* option), or
4. a *UNIX* System Release 5.0 *VAX* Computer or 3B20S Computer archive file (without the *−5* option).

m32convert will transform *infile* to one of the following (respectively):

1. an equivalent *UNIX* System Release 5.0 *VAX* Computer object file or link-edited (a.out) module (with the *−5* option).
2. an equivalent *UNIX* System Release 5.0 *VAX* Computer archive of equivalent object files or link-edited (a.out) modules (with the *−5* option).
3. an equivalent *UNIX* System Release 5.0 archive of unaltered 3B20S Computer object files or link-edited (a.out) modules (with the *−5* option), and
4. an equivalent *VAX* Computer or 3B20S Computer *UNIX* System Release 5.0 portable archive containing unaltered members (without the *−5* option).

All other types of input to the *m32convert* command will be passed unmodified from the input file to the output file (along with appropriate warning messages). When transforming archive files with the *−5* option, the *m32convert* command will inform the user that the archive symbol table has been deleted. To generate an archive symbol table, this archive file must be transformed again by *m32convert* without the *−5* option to create a *UNIX* System Release 5.0 archive file. Then the archive symbol table may be created by executing the *m32ar* command with the *ts* option. If a *UNIX* System Release 5.0 archive with an archive symbol table is being transformed, the archive symbol table will automatically be converted.

FILES

/tmp/conv*

SEE ALSO

m32ar
m32a.out,m32ar

NAME

m32cprs — Compress an Assembler Object File

SYNOPSIS

m32cprs [−**pv**] infile outfile

DESCRIPTION

The **m32cprs** command reduces the size of an assembler object file, *infile*, by removing duplicate structure and union descriptors. The reduced file, *outfile*, is produced as output.

The options are:

−**p** Print statistical messages including: total number of tags, total duplicate tags, and total reduction of *infile*.

−**v** Print verbose error messages if error condition occurs.

EXAMPLE

m32cprs m32a.out sm3b

SEE ALSO

m32strip.

NAME

m32dis – *WE* 32100 Microprocessor Disassembler

SYNOPSIS

m32dis [**–o**][**–V**][**–L**][**–d** sec] [**–da** sec][**–F** function][**–t** sec] [**–l** string] files

DESCRIPTION

The **m32dis** command produces an assembly language listing of each of its object *file* arguments. The listing includes assembly statements and the binary code that produced those statements.

The following *options* are interpreted by the disassembler and may be specified in any order.

–o	Print numbers in octal. Default is hexadecimal.
–V	Version number of the disassembler is written to standard error.
–L	Invokes a lookup of C source labels in the symbol table for subsequent printing.
–d sec	Disassembles the named section as data, printing the offset of the data from the beginning of the section.
–da sec	Disassembles the named section as data, printing the actual address of the data.
–F function	Disassembles single named functions in each object file that is specified on the command line.
–t sec	Disassembles the named section as text.
–l string	Disassemble the library file specified as *string*. For example, one would issue the command **m32dis –l x –lz** to disassemble **libx.a** and **libz.a**. All libraries are assumed to be in **/usr/m32/lib**.

If the **–d, –da** or **–t** options are specified, only those named sections from each user-supplied file name are disassembled. Otherwise, all sections containing text are disassembled. If the **–F** option is specified, only those named functions from each user-supplied filename will be disassembled.

On output, a number enclosed in brackets at the beginning of a line, such as **[5]**, represents that the C breakpointable line number starts with the following instruction. An expression such as **<40>** in the operand field, following a relative displacement for control transfer instructions, is the computed address within the section to which control will be transferred. A C function name appears in the first column, followed by **()**.

DIAGNOSTICS

The self-explanatory diagnostics indicate errors in the command line or problems encountered with the specified files.

SEE ALSO

m32as, m32cc, m32ld.

NAME

m32dump — Dump Selected Parts of an Object File

SYNOPSIS

m32dump [**−acd fghlnoprstuv**] [**−z** name] files

DESCRIPTION

The **m32dump** command dumps selected parts of each of its object *file* arguments.

This command accepts both object files and archives of object files. It processes each file argument according to one or more of the following *options*:

−a Dump the archive header of each member of each archive file argument.

−g Dump the global symbols in the symbol table of an archive.

−f Dump each file header.

−o Dump each optional header.

−h Dump section headers.

−s Dump section contents.

−r Dump relocation information.

−l Dump line number information.

−t Dump symbol table entries.

−z name Dump line number entries for the named function.

−c Dump the string table.

The following *modifiers* are used in conjunction with the *options* listed above to modify their capabilities.

−d number Dump the section number or range of sections starting at *number* and ending either at the last section number or *number* specified by **+d**.

+d number Dump sections in the range either beginning with first section or beginning with section specified by **−d**.

−n name Dump information pertaining only to the named entity. This *modifier* applies to **−h −s, −r, −l,** and **−t**.

−p Suppress printing of the headers.

−t index Dump only the indexed symbol table entry. The **−t** used in conjunction with **+t** specifies a range of symbol table entries.

+t index — Dump the symbol table entries in the range ending with the indexed entry. The range begins at the first symbol table entry or at the entry specified by the **−t** option.

−u — Underline the name of the file for emphasis.

−v — Dump information in symbolic representation rather than numeric (e.g., C_STATIC instead of **0X02**). This *modifier* can be used with the above *options* except **−s** and **−o** options of **m32dump**.

−z name, number — Dump line number entry or range of line numbers starting at *number* for the named function.

+z number — Dump line numbers starting at either function *name* or *number* specified by **−z**, up to *number* specified by **+z**.

Blanks separating an *option* and its *modifier* are optional. The comma separating the name from the number modifying the **−z** option may be replaced by a blank.

The **m32dump** command attempts to format the information it dumps in a meaningful way, printing certain information in character, hex, octal or decimal representation as appropriate.

SEE ALSO

m32a.out, m32ar

NAME

m32ld – Link Editor for *WE* 32100 Microprocessor Object Files

SYNOPSIS

m32ld [**–a**] [**–e** epsym] [**–f** fill] [**–lx**] [**–m**] [**–r**] [**–s**] [**–o** outfile] [**–u** symname] [**–L** dir] [**–N**] [**–V**] [**–VS** num] [**–X**] file–names

DESCRIPTION

The **m32ld** command combines several object files into one, performs relocation, resolves external symbols, and supports symbol table information for symbolic debugging. In the simplest case, the names of several object programs are given, and **m32ld** combines them, producing an object module that can either be executed or used as input for a subsequent **m32ld** run. The output of **m32ld** is left in **m32a.out**. This file is executable if no errors occurred during the load. If any input file, *file–name*, is not an object file, **m32ld** assumes it is either an ASCII file containing link editor directives or an archive library.

If any argument is a library, it is searched exactly once at the point it is encountered in the argument list. Only those routines defining an unresolved external reference are loaded. The library (archive) symbol table is searched sequentially with as many passes as are necessary to resolve external references which can be satisfied by library members. Thus, the ordering of library members is unimportant.

The following options are recognized by **m32ld**.

–a Produce an absolute file; give warnings for undefined references. Relocation information is stripped from the output object file unless the **–r** option is given. The **–r** option is needed only when an absolute file should retain its relocation information (not the normal case). If neither **–a** nor **–r** is given, **–a** is assumed.

–e epsym Set the default entry point address for the output file to be that of the symbol *epsym*. This option forces the **–X** option to be set.

–f fill Set the default fill pattern for "holes" within an output section as well as initialized bss sections. The argument *fill* is a two-byte constant.

–l Specify a library named *x*. It stands for **lib***x***.a** where *x* is up to seven characters. A library is searched when its name is encountered, so the placement of a **–l** is significant. By default, libraries are located in LIBDIR.

–m Generate a map or listing of the input/output sections on the standard output.

−o outfile	Produce an output object file named *outfile*. The name of the default object file is **m32a.out**.
−r	Retain relocation entries in the output object file. Relocation entries must be saved if the output file is to become an input file in a subsequent **m32ld** run. Unless **−a** is also given, the link editor will not complain about unresolved references.
−s	Strip line number entries and symbol table information from the output object file.
−t	Turn off the warning about multiply-defined symbols that are the same size.
−u symname	Enter the argument *symname* as an undefined symbol in the symbol table. This is useful for loading entirely from a library, since initially the symbol table is empty and an unresolved reference is needed to force the loading of the first routine.
−L dir	Change the algorithm of searching for **lib*x*.a** to look in *dir* before looking in LIBDIR.
−m	Output a message for each multiply-defined external definition. However, if the object being loaded include debugging information, extraneous outputs is produced (see the **−g** option in *m32cc*).
−N	Put the data section immediately following the text in the output file.
−V	Output version of **m32ld** being used.
−VS num	Use **num** as a decimal version number identifying the **m32a.out** file that is produced. The version stamp is stored in the optional header.

FILES

File	Description
LIBDIR/lib*x*.a	libraries
m32a.out	output file

CAVEATS

Through its input directives, the link editor gives users great flexibility; however, people who use the input directives must assume some added responsibilities. Input directives should insure the following properties for programs:

C defines a zero pointer as null. A pointer to which zero has been assigned must not point to any object. To satisfy this, users must not place any object at virtual address zero in the data space.

NAME

m32list – Produce C Source Listing from *WE* 32100 Microprocessor Object File

SYNOPSIS

m32list [–V] [–h] source file ...[object-file]

DESCRIPTION

The **m32list** command produces a C source listing with line number information. If multiple C source files were used to create the object file, **m32list** will accept multiple file names. The object file is taken to be the last non-C source file argument. If no object file is specified, the **m32a.out** default object file, **m32.out** is used.

Line numbers are printed for each breakpoint inserted by the compiler (generally, each executable C statement that begins a new line of source code). Line numbering begins at once for each function. Line number 1 is always the line containing the left curly brace ({) that begins the function body. Line numbers are also supplied for inner block redeclarations of local variables so that they can be distinguished by the symbolic debugger.

The –V flag supplies **m32list** version information.

The –**h** flag suppresses heading output.

CAVEATS

Object files given to **m32list** must have symbolic debugging symbols.

Since **m32list** does not use the C preprocessor, it may be unable to recognize function definitions whose syntax has been distorted by the use of C preprocessor macro substitutions.

SEE ALSO

m32as, **m32cc**, **m32ld**.

DIAGNOSTICS

m32list will produce the error message.
m32list: name: cannot open if *name* cannot be read.

The following messages are produced when **m32list** has become confused by **#ifdef**'s in the source file:

m32list: name: out of synch: too many }
m32list: name: unexpected end-of-file

The error message

m32list: name: missing or inappropriate line numbers

means that either symbolic debugging information is missing or **m32list** has been confused by C preprocessor statements.

If the source file names do not end in .c the message is

m32list: name: invalid C source name

An invalid object file will cause the message

m32list: name: bad magic

to be produces. If some or all of the symbolic debugging information is missing, one of the following messages will be printed:

m32list: name: symbols have been stripped, cannot proceed
m32list: name: cannot read line numbers
m32list: name: not in symbol table

NAME

m32lorder — Find Ordering Relation for an Object Library

SYNOPSIS

m32lorder files

DESCRIPTION

The input is one or more object or library archive *files*. The standard output is a list of pairs of object file names, meaning that the first file of the pair refers to external identifiers defined in the second.

The output may be processed by **tsort(1)** to find a ordering of a library suitable for one-pass access by **m32ld(1)**. The link editor is capable of multiple passes over the archive and does not require that **m32lorder** be used when building an archive. The usage of **m32lorder** may, however, allow for a slightly more efficient access of the archive during the link edit process.

The following example builds a new library from existing .o files.

```
ar cr library 'm32lorder *.o | tsort'
```

FILES

*symref, *symdef temporary files

SEE ALSO

m32ld, m32ar, sort

BUGS

Object files whose names do not end with **.o**, even when contained in library archives, are overlooked. Their global symbols and references are attributed to some other file.

NAME

m32man — Print On-Line Documentation for *WE* 32100 Microprocessor

SYNOPSIS

m32man *command*

DESCRIPTION

m32man is a shell command file which prints on-line documentation for *WE* 32100 Microprocessor commands.

DIAGNOSTICS

can't open MANDIR/*command*.out
Manual page for command is not on system.

FILES

MANDIR/*command*.out

NAME

m32nm – Print Name List of *WE* 32100 Microprocessor Object File

SYNOPSIS

m32nm [**–o**][**–x**] [**–v**] [**–n**] [**–e**] [**–f**] [**–u**] [**–V**] file name...

DESCRIPTION

The **m32nm** command displays the symbol table of each object file *file-name*. *file-name* may be a relocatable or absolute object file or it may be an archive of such object files. For each symbol, the following information is printed:

Name The name of the symbol.

Value Its value expressed as an offset or an address depending on its storage class.

Class Its storage class.

Type Its type and derived type. If the symbol is an instance of a structure or of a union then the structure or union tag is given following the type (e.g., struct-tag). If the symbol is an array, then the array dimensions are given following the type (e.g., **char[n][m]**).

Size Its size in bytes, if available.

Line The source line number at which it is defined, if available.

Section For storage classes static and external, the object file section containing the symbol.

The output of **m32nm** may be controlled using the following flags:

–o A symbol's value and size are printed in octal instead of decimal.

–x A symbol's value and size are printed in hexadecimal instead of decimal.

–h Do not display the output header data.

–v External symbols are sorted by value before being printed.

–n External symbols are sorted by name before being printed.

–e Only static and external symbols are printed.

–f "Fancy" output is produced; that is, the symbol table information is post-processed to reflect the block structure of the source code.

–u Only undefined symbols are printed.

–r Prepend the name of the object file to each output line.

−p Produce easily parsed, terse output. Each symbol name is preceded by its value (blanks if undefined) and one of the letters U (undefined), A (absolute), T (text segment symbol), D (data segment symbol), S (user-defined segment symbol), R (register symbol), F (file symbol), or C (common symbol). If the symbol is local (nonexternal), the type letter is in lower case.

−V Print the version of **nm** command executing on the standard error output.

−T By default, **nm** prints the entire name of the symbols listed. Since object files can have symbol names with an arbitrary number of characters, a name that is longer than the width of the column set aside for names will overflow its column, forcing every column after the name to be misaligned. The **−T** option causes **nm** to truncate every name which would otherwise overflow its column and place an asterisk as the last character in the displayed name to mark it as truncated.

Flags may be used in any order, either singly or in combination, and may appear anywhere in the command line. Therefore, both **m32nm name −e −v** and **m32nm −ve name** print the static and external symbols in **name**, with external symbols sorted by value.

FILES

/usr/tmp/nm??????

SEE ALSO

m32as,m32cc,m32ld.

DIAGNOSTICS

m32nm: name: cannot open
if name cannot be read.

m32nm: name: bad magic
if name is not an object file.

m32nm: name: no symbols
if the symbols have been stripped from name.

NAME

m32size — Print Section Sizes for *WE* 32100 Microprocessor Object Files

SYNOPSIS

m32size [−o] [−x] [−V] files

DESCRIPTION

The **m32size** command produces section size information for each section in the object files.

Numbers are printed in decimal unless either the **−o** or the **−x** option is used, in which case they are printed in octal or in hexadecimal, respectively.

The **−V** flag supplies version information on the **m32size** command.

SEE ALSO

m32as, m32cc, m32ld.

DIAGNOSTICS

m32size: name: cannot open
if *name* cannot be read.

m32size: name: bad magic
if *name* is not a *WE* 32100 Microprocessor object file.

NAME

m32strip – Strip Symbol and Line Number Information From *WE* 32100 Microprocessor Object File

SYNOPSIS

m32strip [–l] [–x] [–r] [–s] [–V] file-names

DESCRIPTION

The **m32strip** command strips the symbol table and line number information from object files, including archives. Once this has been done, no symbolic debugging access is available for that file; therefore, this command is normally run only on production models that have been debugged and tested.

The amount of information stripped from the symbol table can be controlled using the following options:

–l Strip line number information only; do not strip any symbol table information.

–x Do not strip static or external symbol information.

–r Reset the relocation indices into the symbol table.

–b Same as the **–x** option, but also do not strip scoping information (i.e., beginning and end of block delimiters).

–V Print version of **m32strip** command executing.

If there are any relocation entries in the object file and any symbol table information is to be stripped, **m32strip** will terminate without stripping *file-name* unless the **–r** flag is used.

If the **m32strip** command is executed on a common archive file (see **m32ar** file format), the archive symbol table will be removed. The archive symbol table must be restored by executing the **m32ar** command with the **s** option before the archive can be link-edited by the **ld** command. **m32strip** will instruct the user with appropriate warning messages when this instruction arises.

The purpose of this command is to reduce the file storage overhead taken by the object file.

FILES

/usr/tmp/m32str??????

SEE ALSO

m32as, **m32cc**, **m32ld**

DIAGNOSTICS

m32strip: name: cannot open

m32strip: name: bad magic
if *name* is not a *WE* 32100 Microprocessor object file

m32strip: name relocation entries present; cannot strip
if *name* contains relocation entries, the **−r** flag not used, and any symbol table information was to be stripped.

NAME

a64l, l64a — convert between long integer and base-64 ASCII string

SYNOPSIS

long a64l (s)
char *s;

char *l64a (l)
long l;

DESCRIPTION

These functions are used to maintain numbers stored in base-64 ASCII characters. This is a notation by which long integers can be represented by up to six characters; each character represents a "digit" in a radix-64 notation.

The characters used to represent "digits" are **.** for 0, **/** for 1, **0** through **9** for 2—11, **A** through **Z** for 12—37, and **a** through **z** for 38—63.

a64l takes a pointer to a null-terminated base-64 representation and returns a corresponding **long** value. If the string pointed to by *s* contains more than six characters, *a64l* will use the first six.

l64a takes a **long** argument and returns a pointer to the corresponding base-64 representation. If the argument is 0, *l64a* returns a pointer to a null string.

BUGS

The value returned by *l64a* is a pointer into a static buffer, the contents of which are overwritten by each call.

NAME

abs — return integer absolute value

SYNOPSIS

int abs (i)
int i;

DESCRIPTION

abs returns the absolute value of its integer operand.

BUGS

In two's complement representation, the absolute value of the negative integer with largest magnitude is undefined. Some implementations trap this error, but others simply ignore it.

SEE ALSO

floor(3M).

NAME

bsearch — binary search a sorted table

SYNOPSIS

#include <search.h>

char *bsearch ((char *) key, (char *) base, nel, sizeof (*key), compar)
unsigned nel;
int (*compar)( );

DESCRIPTION

bsearch is a binary search routine generalized from Knuth (6.2.1) Algorithm B. It returns a pointer into a table indicating where a datum may be found. The table must be previously sorted in increasing order according to a provided comparison function. *key* points to a datum instance to be sought in the table. *base* points to the element at the base of the table. *nel* is the number of elements in the table. *compar* is the name of the comparison function, which is called with two arguments that point to the elements being compared. The function must return an integer less than, equal to, or greater than zero as accordingly the first argument is to be considered less than, equal to, or greater than the second.

EXAMPLE

The example below searches a table containing pointers to nodes consisting of a string and its length. The table is ordered alphabetically on the string in the node pointed to by each entry.

This code fragment reads in strings and either finds the corresponding node and prints out the string and its length, or prints an error message.

```
#include <stdio.h>
#include <search.h>

#define TABSIZE     1000

struct node {                    /* these are stored in the table */
        char *string;
        int length;
};
struct node table[TABSIZE];      /* table to be searched */
        .
        .
        .
{
        struct node *node_ptr, node;
        int node_compare ( ); /* routine to compare 2 nodes */
        char str_space[20];   /*space to read string into */
```

```
.
.
.
node.string = str_space;
while (scanf("%s", node.string) != EOF) {
        node_ptr = (struct node *)bsearch((char *)(node),
                    (char *)table, TABSIZE,
                    sizeof(struct node), node_compare);
        if (node_ptr != NULL) {
                (void)printf("string = %20s, length = %d\n",
                    node_ptr->string, node_ptr->length);
        } else {
                (void)printf("not found: %s\n", node.string);
                }
        }
}
/*
        This routine compares two nodes based on an
        alphabetical ordering of the string field.
*/
int
node_compare(node1, node2)
struct node *node1, *node2;
{
        return strcmp(node1->string, node2->string);
}
```

NOTES

The pointers to the key and the element at the base of the table should be of type pointer-to-element, and cast to type pointer-to-character. The comparison function need not compare every byte, so arbitrary data may be contained in the elements in addition to the values being compared. Although declared as type pointer-to-character, the value returned should be cast into type pointer-to-element.

SEE ALSO

hsearch(3C), lsearch, qsort(3C), tsearch(3C).

DIAGNOSTICS

A NULL pointer is returned if the key cannot be found in the table.

NAME

toupper, tolower, _toupper, _tolower, toascii — translate characters

SYNOPSIS

```
#include <ctype.h>
int toupper (c)
int c;
int tolower (c)
int c;
int _toupper (c)
int c;
int _tolower (c)
int c;
int toascii (c)
int c;
```

DESCRIPTION

toupper and *tolower* have as domain the range of *getc(3S)*: the integers from −1 through 255. If the argument of *toupper* represents a lower-case letter, the result is the corresponding upper-case letter. If the argument of *tolower* represents an upper-case letter, the result is the corresponding lower-case letter. All other arguments in the domain are returned unchanged.

The macros *_toupper* and *_tolower* are macros that accomplish the same thing as *toupper* and *tolower* but have restricted domains and are faster. *_toupper* requires a lower-case letter as its argument; its result is the corresponding upper-case letter. The macros *_tolower* requires an upper-case letter as its argument; its result is the corresponding lower-case letter. Arguments outside the domain cause undefined results.

toascii yields its argument with all bits turned off that are not part of a standard ASCII character; it is intended for compatibility with other systems.

SEE ALSO

ctype, getc(3S).

NAME

crypt, setkey, encrypt — generate DES encryption

SYNOPSIS

char *crypt (key, salt)
char *key, *salt;

void setkey (key)
char *key;

void encrypt (block, edflag)
char *block;
int edflag;

DESCRIPTION

crypt is the password encryption function. It is based on the NBS Data Encryption Standard (DES), with variations intended (among other things) to frustrate use of hardware implementations of the DES for key search.

key is a user's typed password. *salt* is a two-character string chosen from the set **[a-zA-Z0-9./]**; this string is used to perturb the DES algorithm in one of 4096 different ways, after which the password is used as the key to encrypt repeatedly a constant string. The returned value points to the encrypted password. The first two characters are the salt itself.

The *setkey* and *encrypt* entries provide (rather primitive) access to the actual DES algorithm. The argument of *setkey* is a character array of length 64 containing only the characters with numerical value 0 and 1. If this string is divided into groups of 8, the low-order bit in each group is ignored; this gives a 56-bit key which is set into the machine. This is the key that will be used with the above mentioned algorithm to encrypt or decrypt the string *block* with the function *encrypt*.

The argument to the *encrypt* entry is a character array of length 64 containing only the characters with numerical value 0 and 1. The argument array is modified in place to a similar array representing the bits of the argument after having been subjected to the DES algorithm using the key set by *setkey*. If *edflag* is zero, the argument is encrypted; if non-zero, it is decrypted.

SEE ALSO

login(1), passwd(1), getpass(3C), passwd(4).

BUGS

The return value points to static data that are overwritten by each call.

NAME

isalpha, isupper, islower, isdigit, isxdigit, isalnum, isspace, ispunct, isprint, isgraph, iscontrl, isascii — classify characters

SYNOPSIS

#include <ctype.h>
int isalpha (c)
int c;
...

DESCRIPTION

These macros classify character-coded integer values by table lookup. Each is a predicate returning nonzero for true, zero for false. *isascii* is defined on all integer values; the rest are defined only where *isascii* is true and on the single non-ASCII value **EOF** (−1 — see *stdio(3S)*).

isalpha *c* is a letter.

isupper *c* is an upper-case letter.

islower *c* is a lower-case letter.

isdigit *c* is a digit [0—9].

isxdigit *c* is a hexadecimal digit [0—9],[A—F] or [a—f].

isalnum *c* is an alphanumeric (letter or digit).

isspace *c* is a space, tab, carriage return, new-line, vertical tab, or form-feed.

ispunct *c* is a punctuation character (neither control nor alphanumeric).

isprint *c* is a printing character, code 040 (space) through 0176 (tilde).

isgraph *c* is a printing character, like *isprint* except false for space.

iscntrl *c* is a delete character (0177) or an ordinary control character (less than 040).

isascii *c* is an ASCII character, code less than 0200.

DIAGNOSTICS

If the argument to any of these macros is not in the domain of the function, the result is undefined.

SEE ALSO

stdio(3S), ascii(5).

NAME

l3tol ltol3 — convert between 3-byte integers and long integers

SYNOPSIS

```
void l3tol (lp, cp, n)
long *lp;
char *cp;
int n;

void ltol3 (cp, lp, n)
char *cp;
long *lp;
int n;
```

DESCRIPTION

l3tol converts a list of *n* three-byte integers packed into a character string pointed to by *cp* into a list of long integers pointed to by *lp*.

ltol3 performs the reverse conversion from long integers (*lp*) to three-byte integers (*cp*).

These functions are useful for file-system maintenance where the block numbers are three bytes long.

SEE ALSO

fs(4).

BUGS

Because of possible differences in byte ordering, the numerical values of the long integers are machine-dependent.

NAME

ldahread — Read Archive Header of an Archive File Member

SYNOPSIS

```
#include <stdio.h>
#include <ar.h>
#include "INCDIR/filehdr.h"
#include "INCDIR/ldfcn.h"

int ldahread (ldptr,arhead)
LDFILE *ldptr;
ARCHDR *arhead;
```

DESCRIPTION

If **TYPE**(*ldptr*) is the archive file magic number, *idahread* reads the archive header of the object file currently associated with *ldptr* into the area of memory beginning at *arhead*.

ldahread returns **SUCCESS** or **FAILURE**. *ldahread* fails if **TYPE**(*ldptr*) does not represent an archive file, or if it cannot read the archive header.

The program must be loaded with the object file access routine library **LIBDIR/libld.a.**

intro describes *LIBDIR* and *INCDIR*.

SEE ALSO

idclose, *ldopen*, *ldfcn*, *m32ar* format

NAME

ldclose, ldaclose — Close a *WE* 32100 Microprocessor Object File

SYNOPSIS

```
#include <stdio.h>
#include "INCDIR/filehdr.h"
#include "INCDIR/ldfcn.h"

int ldclose (ldptr)
LDFILE *ldptr;

int ldaclose (ldptr)
LDFILE *ldptr;
```

DESCRIPTION

ldopen and *idclose* provide uniform access to both simple object files and object files that are members of archive files. Thus an archive of object files can be processed as if it were a series of simple object files.

If **TYPE**(*ldptr*) does not represent an archive file, *idclose* will close the file and free the memory allocated to the **LDFILE** structure associated with *ldptr*. If **TYPE**(*ldptr*) is the magic number of an archive file, and if there are any more files in the archive, *idclose* reinitializes **OFFSET**(*ldptr*) to the file address of the next archive member and returns **FAILURE**. The **LDFILE** structure is prepared for a subsequent *ldopen*. In all other cases, *ldclose* returns **SUCCESS**.

ldaclose closes the file and frees the memory allocated to the **LDFILE** structure associated with ldptr regardless of the value of **TYPE**(*ldptr*). *ldaclose* always returns **SUCCESS**. The function is often used in conjunction with *idaopen*.

The program must be loaded with the object file access routine library **LIBDIR/libld.a.**

intro describes *INCDIR* and *LIBDIR*.

SEE ALSO

idopen, intro, idfcn, paths

NAME

ldfhread — Read the File Header for a *WE* 32100 Microprocessor Object File

SYNOPSIS

```
#include <stdio.h>
#include "INCDIR/filehdr.h"
#include "INCDIR/ldfcn.h"

int ldfhread (ldptr, filehead)
LDFILE *ldptr;
FILHDR *filehead;
```

DESCRIPTION

ldfhread reads the file header of the object file currently associated with *ldptr* into the area of memory beginning at *filehead*.

ldfhread returns **SUCCESS** or **FAILURE**. *ldfhread* fails if it cannot read the file header.

In most cases the use of *ldfhread* can be avoided by using the macro **HEADER**(*ldptr*) defined in **ldfcn.h** (see *ldfcn*). The information in any field, *fieldname*, of the file header may be accessed using **HEADER**(*ldptr*).*fieldname*.

The program must be loaded with the object file access routine library **LIBDIR/libld.a**.

intro describes *INCDIR* and *LIBDIR*.

SEE ALSO

ldclose, ldopen, intro, ldfcn, paths

NAME

ldgetname — retrieve symbol name for object file symbol table entry

SYNOPSIS

```
#include <stdio.h>
#include "INCDIR/filehdr.h"
#include "INCDIR/syms.h"
#include "INCDIR/ldfcn.h"

char *ldgetname (ldptr, symbol)
LDFILE *ldptr;
SYMENT *symbol;
```

DESCRIPTION

Ldgetname returns a pointer to the name associated with *symbol* as a string. The string is contained in a static buffer local to *ldgetname* that is overwritten by each call to *ldgetname*, and therefore must be copied by the caller if the name is to be saved.

Ldgetname will return NULL (defined in **stdio.h**) for an object file if the name cannot be retrieved. This situation can occur:

- if the "string table" cannot be found,
- if enough memory cannot be allocated for the string table,
- if the string table appears not to be a string table (for example, if an auxiliary entry is handed to *ldgetname* that looks like a reference to a name in a nonexistent string table), or
- if the name's offset into the string table is past the end of the string table.

Typically, *ldgetname* will be called immediately after a successful call to *ldtbread* to retrieve the name associated with the symbol table entry filled by *ldtbread.*

The program must be loaded with the object file access routine library LIBDIR/libld.a.

intro describes *INCDIR* and *LIBDIR.*

SEE ALSO

ldclose, ldopen, ldtbseek, ldtbread, intro ldfcn, paths

NAME

ldlread, ldlinit, ldlitem — Manipulate Line Number Entries for a *WE* 32100 Microprocessor Object File Function

SYNOPSIS

```
#include<stdio.h>
#include "INCDIR/filehdr.h"
#include "INCDIR/linenum.h"
#include "INCDIR/ldfcn.h"

int ldlread(ldptr,fcnindx,linenum,linent)
LDFILE *ldptr;
long fcnindx;
unsigned short linenum;
LINENO linent;

int ldlinit(ldptr,fcnindx)
LDFILE *ldptr;
long fcnindx;

int ldlitem(ldptr, linenum, linent)
LDFILE *ldptr;
unsigned short linenum;
LINENO linent;
```

DESCRIPTION

ldlread searches the line number entries of the object file currently associated with *ldptr*. *ldlread* begins its search with the line number entry for the beginning of a function and confines its search to the line numbers associated with a single function. The function is identified by *fcnindx*, the index of its entry in the object file symbol table. *ldlread* reads the entry with the smallest line number equal to or greater than *linenum* into *linent*.

ldlinit and *ldlitem* together perform exactly the same function as *ldlread*. After an initial call to *ldlread* or *ldlinit*, *ldlitem* may be used to retrieve a series of line number entries associated with a single function. *ldlinit* simply locates the line number entries for the function identified by *fcnindx*. *ldlitem* finds and reads the entry with the smallest line number equal to or greater than *linenum* into *linent*.

ldlread, ldlinit, and *ldlitem* each return either **SUCCESS** or **FAILURE**. *ldlread* fails if there are no line number entries in the object file, if *fcnindx* does not index a function entry in the symbol table, or if it finds no line number equal to or greater than *linenum*. *ldlinit* fails if there are no line number entries in the object file or if *fcnindx* does not index a function entry in the symbol table. *ldlitem* fails if it finds no line number equal to or greater than *linenum*.

The programs must be loaded with the object file access routine library *LIBDIR/libld.a*.

SEE ALSO

ldclose, ldopen, ldtbindex, intro, ldfcn, paths.

NAME

lsearch, lfind — linear search and update

SYNOPSIS

```
#include <stdio.h>
#include <search.h>

char *lsearch ((char *)key, (char *)base, nelp, sizeof(*key), compar)
unsigned *nelp;
int (*compar)( );

char *lfind ((char *)key, (char *)base, nelp, sizeof(*key), compar)
unsigned *nelp;
int (*compar)( );
```

DESCRIPTION

lsearch is a linear search routine generalized from Knuth (6.1) Algorithm S. It returns a pointer into a table indicating where a datum may be found. If the datum does not occur, it is added at the end of the table. *key* points to the datum to be sought in the table. *base* points to the first element in the table. *nelp* points to an integer containing the current number of elements in the table. The integer is incremented if the datum is added to the table. *compar* is the name of the comparison function which the user must supply (*strcmp*, for example). It is called with two arguments that point to the elements being compared. The function must return zero if the elements are equal and nonzero otherwise.

lfind is the same as *lsearch* except that if the datum is not found, it is not added to the table. Instead, a NULL pointer is returned.

NOTES

The pointers to the key and the element at the base of the table should be of type pointer-to-element, and cast to type pointer-to-character. The comparison function need not compare every byte, so arbitrary data may be contained in the elements in addition to the values being compared. Although declared as type pointer-to-character, the value returned should be cast into type pointer-to-element.

EXAMPLE

This fragment will read in ≤ TABSIZE strings of length ≤ ELSIZE and store them in a table, eliminating duplicates.

```
#include <stdio.h>
#include <search.h>

#define TABSIZE 50
#define ELSIZE 120
```

```
char line[ELSIZE], tab[TABSIZE][ELSIZE], *lsearch( );
unsigned nel = 0;
int strcmp( );
...

while (fgets(line, ELSIZE, stdin) != NULL &&
  nel < TABSIZE)
    (void) lsearch(line, (char *)tab, &nel,
            ELSIZE, strcmp);
...
```

SEE ALSO

bsearch, hsearch(3C), tsearch(3C).

DIAGNOSTICS

If the searched for datum is found, both *lsearch* and *lfind* return a pointer to it. Otherwise, *lfind* returns NULL and *lsearch* returns a pointer to the newly added element.

BUGS

Undefined results can occur if there is not enough room in the table to add a new item.

NAME

ldlseek,idnlseek — Seek to Line Number Entries of a Section of a *WE* 32100 Microprocessor Object File

SYNOPSIS

```
#include <stdio.h>
#include "INCDIR/filehdr.h"
#include "INCDIR/ldfcn.h"

int ldlseek (ldptr, sectindx)
LDFILE *ldptr;
unsigned short sectindx;

int ldnlseek (ldptr, sectname)
LDFILE *ldptr;
char *sectname;
```

DESCRIPTION

ldlseek seeks to the line number entries of the section specified by *sectindx* of the object file currently associated with *ldptr*.

ldnlseek seeks to the line number entries of the section specified by *sectname*.

ldlseek and *ldnlseek* return **SUCCESS** or **FAILURE**. *ldlseek* fails if *sectindx* is greater than the number of sections in the object file; *ldnlseek* fails if there is no section name corresponding with **sectname*. Either function fails if the specified section has no line number entires or if it cannot seek to the specified line number entries.

Note that the first section has an index of *one*.

The program must be loaded with the object file access routine library **LIBDIR/libld.a**.

intro describes *INCDIR* and *LIBDIR*.

SEE ALSO

ldclose, ldopen, ldshread, intro, ldfcn, paths

NAME

ldohseek — Seek to the Optional File Header of a *WE* 32100 Microprocessor Object File

SYNOPSIS

```
#include <stdio.h>
#include "INCDIR/filehdr.h"
#include "INCDIR/ldfcnh.h"

int ldohseek (ldptr)
LDFILE *ldptr;
```

DESCRIPTION

ldohseek seeks to the optional file header of object file currently associated with *ldptr*.

ldohseek returns **SUCCESS** or **FAILURE**. *ldohseek* fails if the object file has no optional header or if it cannot seek to the optional header.

The program must be loaded with the object file access routine library **LIBDIR**/libld.a.

intro describes *INCDIR* and *LIBDIR*.

SEE ALSO

ldclose, ldopen, ldfhread, intro, ldfcn, paths

NAME

ldopen, ldaopen — Open a *WE* 32100 Microprocessor File for Reading

SYNOPSIS

```
#include <stdio.h>
#include "INCDIR/filehdr.h"
#include "INCDIR/ldfcn.h"

LDFILE *ldopen (filename, ldptr)
char *filename;
LDFILE *ldptr;

LDFILE *ldaopen (filename, oldptr)
char *filename;
LDFILE *oldptr;
```

DESCRIPTION

ldopen and *ldclose* provide uniform access to both simple object files and object files that are members of archive files. Thus an archive of object files can be processed as if it were a series of simple object files.

If *ldptr* has the value **NULL**, then *ldopen* opens *filename* and allocates and initializes the **LDFILE** structure, and returns a pointer to the structure to the calling program.

If *ldptr* is valid and if **TYPE**(*ldptr*) is the archive magic number, *ldopen* reinitializes the **LDFILE** structure for the next archive member of *filename*.

ldopen and *ldclose* are designed to work in concert. *ldclose* returns **FAILURE** only when **TYPE**(*ldptr*) is the archive magic number and there is another file in the archive to be processed. Only then should *ldopen* be called with the current value of *ldptr*. In all other cases, in particular whenever a new *filename* is opened, *ldopen* should be called with a **NULL** *ldptr* argument.

The following is a prototype for the use of *ldopen* and *ldclose*.

```
/*for each filename to be processed*/
ldptr = NULL;
do
        if (ldptr + ldopen(filename, ldptr)) != NULL)
        {
            /* check magic number */
            /* process the file */
        }
}while (ldclose(ldptr) == FAILURE);
```

If the value of *oldptr* is not **NULL**, *ldaopen* opens *filename* anew and allocates and initializes a new **LDFILE** structure, copying the **TYPE**, **OFFSET**, and **HEADER** fields from *oldptr*. *ldaopen* returns a pointer to the new **LDFILE** structure. This new pointer is independent of the old pointer, *oldptr*. The two pointers may be used concurrently to read separate parts of the object file. For example, one pointer may be used to step sequentially through the relocation information, while the other is used to read indexed symbol table entries.

Both *ldopen* and *ldaopen* open *filename* for reading. Both functions return **NULL** if *filename* cannot be opened, or if memory for the *LDFILE* structure cannot be allocated. A successful open does not insure that the given file is an object file or an archived object file.

The program must be loaded with the object file access routine library **LIBDIR/libld.a**.

intro describes *INCDIR* and *LIBDIR*.

SEE ALSO

ldclose, intro, ldfcn, paths

NAME

ldrseek,ldnrseek — Seek to Relocation Entries of a Section of a *WE* 32100 Microprocessor Object File

SYNOPSIS

```
#include <stdio.h>
#include "INCDIR/filehdr.h"
#include "INCDIR/ldfcn.h"

int ldrseek (ldptr, sectindx)
LDFILE *ldptr;
unsigned short sectindx;

int ldnrseek (ldptr, sectname)
LDFILE *ldptr;
char *sectname;
```

DESCRIPTION

ldrseek seeks to the relocation entries of the section specified by *sectindx* of the object file currently associated with *ldptr*.

ldnrseek seeks to the relocation entries of the section specified by *sectname*.

ldrseek and *ldnrseek* return **SUCCESS** or **FAILURE**. *ldrseek* fails if *sectindx* is greater than the number of sections in the object file; *ldnrseek* fails if there is no section name corresponding with *sectname*. Either function fails if the specified section has no relocation entries or if it cannot seek to the specified relocation entries.

Note that the first section has an index of *one*.

The program must be loaded with the object file access routine library *LIBDIR*/**libld.a**.

intro describes INCDIR and *LIBDIR*.

SEE ALSO

ldclose, ldopen, ldshread, intro, ldfcn, paths

NAME

ldshread,ldnshread — Read an Indexed/Named Section Header of a *WE* 32100 Microprocessor Object File

SYNOPSIS

```
#include <stdio.h>
#include "INCDIR/filehdr.h"
#include "INCDIR/scnhdr.h"
#include "INCDIR/ldfcn.h"

int ldshread (ldptr, sectindx, secthead)
LDFILE *ldptr;
unsigned short sectindx;
SCNHDR *secthead;

int ldnshread (ldptr, sectname, secthead)
LDFILE *ldptr;
char sectname;
SCNHDR *secthead;
```

DESCRIPTION

ldshread reads the section header specified by *sectindx* of the object file currently associated with *ldptr* into the area of memory beginning at *secthead*.

ldnshread reads the section header specified by *sectname* into the area of memory beginning at *secthead*.

ldshread and *ldnshread* return **SUCCESS** or **FAILURE**. *ldshread* fails if *sectindx* is greater than the number of sections in the object file; *ldnshread* fails if there is no section name corresponding with *sectname*. Either function fails if it cannot read the specified section header.

Note that the first section header has an index of *one*.

The program must be loaded with the object file access routine library **LIBDIR/libld.a**.

intro describes *INCDIR* and *LIBDIR*.

SEE ALSO

ldclose, ldopen, intro, ldfcn, paths

NAME

ldsseek, ldnsseek — Seek to an Indexed/Named Section of a *WE* 32100 Microprocessor Object File

SYNOPSIS

```
#include <stdio.h>
#include "INCDIR/filehdr.h"
#include "INCDIR/ldfcn.h"

int ldsseek (ldptr, sectindx)
LDFILE *ldptr;
unsigned short sectindx;

int ldnsseek (ldptr, sectname)
LDFILE *ldptr;
char *sectname;
```

DESCRIPTION

ldsseek seeks to the section specified by *sectindx* of the object file currently associated with *ldptr*.

ldnsseek seeks to the section specified by *sectname*.

ldsseek and *ldnsseek* return **SUCCESS** or **FAILURE**. *ldsseek* fails if *sectindx* is greater than the number of sections in the object file; *ldnsseek* fails if there is no section name corresponding with *sectname*. Either function fails if there is no section data for the specified section or if it cannot seek to the specified section.

Note that the first section has an index of *one*.

The program must be loaded with the object file access routine library **LIDIR/libld.a**.

intro describes *INCDIR* and *LIBDIR*.

SEE ALSO

ldclose, ldopen, ldshread, intro, ldfcn, paths

NAME

ldtbindex — Compute the Index of a Symbol Table Entry of a *WE* 32100 Microprocessor Object File

SYNOPSIS

```
#include <stdio.h>

#include "INCDIR/filehdr.h"
#include "INCDIR/syms.h"
#include "INCDIR/ldfcn.h"

long ldtbindex (ldpr)
LDFILE *ldptr;
```

DESCRIPTION

ldtbindex returns the (long) index of the symbol table entry at the current position of the object file associated with *ldpr*.

The index returned by *ldtbindex* may be used in subsequent calls to *ldtbread*. However, since *ldtbindex* returns the index of the symbol table entry that begins at the current position of the object file, if *ldtbindex* is called immediately after a particular symbol table entry has been read, it returns the index of the next entry.

ldtbindex fails if there are no symbols in the object file, or if the object file is not positioned at the beginning of a symbol table entry.

Note that the first symbol in the symbol table has an index of *zero*.

The program must be loaded with the object file access routine library **LIBDIR/libld.a**.

intro describes *INCDIR* and *LIBDIR*.

SEE ALSO

ldclose, ldopen, ldtbread, ldtbseek, intro, ldfcn, paths

NAME

ldtbread — Read an Indexed Symbol Table Entry of a *WE* 32100 Microprocessor Object File

SYNOPSIS

```
#include <stdio.h>
#include "INCDIR/filehdr.h"
#include "INCDIR/syms.h"
#include "INCDIR/ldfcn.h"

int ldtbread (ldptr, symindex, symbol)
LDFILE *ldptr;
long symindex;
SYMENT *symbol;
```

DESCRIPTION

ldtbread reads the symbol table entry specified by *symindex* of the object file currently associated with *ldptr* into the area of memory beginning at *symbol*.

ldtbread returns **SUCCESS** or **FAILURE**. *ldtbread* fails if **symindex** is greater than the number of symbols in the object file, or if it cannot read the specified symbol table entry.

Note that the first symbol in the symbol table has an index of *zero*.

The program must be loaded with the object file access routine library **LIBDIR/libld.a**.

intro describes *INCDIR* and *LIBDIR*.

SEE ALSO

ldclose, ldopen, ldtbseek, intro, ldfcn, paths

NAME

ldtbseek — Seek to the Symbol Table of a *WE* 32100 Microprocessor Object File

SYNOPSIS

```
#include <stdio.h>
#include "INCDIR/filehdr.h"
#include "INCDIR/ldfcnh.h"

int ldtbseek (ldptr)
LDFILE *ldptr
```

DESCRIPTION

ldtbseek seeks to the symbol table of the object file currently associated with *ldptr*.

ldtbseek return **SUCCESS** or **FAILURE**. *ldtbseek* fails if the symbol table has been stripped from the object file, or if it cannot seek to the symbol table.

The program must be loaded with the object file access routine library **LIBDIR/libld.a**.

intro describes *INCDIR* and *LIBDIR*.

SEE ALSO

ldclose, ldopen, ldtbread, intro, ldfcn, paths

NAME

memccpy, memchr, memcmp, memcpy, memset — memory operations

SYNOPSIS

```
#include <memory.h>
char *memccpy (s1, s2, c, n)
char *s1, *s2;
int c, n;
char *memchr (s, c, n)
char *s;
int c, n;
int memcmp (s1, s2, n)
char *s1, *s2;
int n;
char *memcpy (s1, s2, n)
char *s1, *s2;
int n;
char *memset (s, c, n)
char *s;
int c, n;
```

DESCRIPTION

These functions operate as efficiently as possible on memory areas (arrays of characters bounded by a count, not terminated by a null character). They do not check for the overflow of any receiving memory area.

memccpy copies characters from memory areas **s2** into **s1**, stopping after the first occurrence of character **c** has been copied, or after **n** characters have been copied, whichever comes first. It returns a pointer to the character after the copy of **c** in **s1**, or a NULL pointer if **c** was not found in the first **n** characters of **s2**.

memchr returns a pointer to the first occurrence of character **c** in the first **n** characters of memory areas **s**, or a NULL pointer if **c** does not occur.

memcmp compares its arguments, looking at the first **n** characters only, and returns an integer less than, equal to, or greater than 0, according as **s1** is lexicographically less than, equal to, or greater than **s2**.

memcpy copies **n** characters from memory area **s2** to **s1**. It returns **s1**.

memset sets the first **n** characters in memory area **s** to the value of character **c**. It returns **s**.

NOTE

For user convenience, all these functions are declared in the optional *<memory.h>* header file.

BUGS

memcmp uses native character comparison, which is signed on *PDP* 11 Computers and *VAX* 11 Computers, unsigned on other machines. Thus the sign of the value returned when one of the characters has its high-order bit set is implementation-dependent.

Character movement is performed differently in different implementations. Thus overlapping moves may yield surprises.

NAME

printf, sprintf — print formatted output

SYNOPSIS

#include <stdio.h>

int printf (format [, arg] ...)
char *format;

int sprintf (s, format [, arg] ...)
char *s format;

DESCRIPTION

printf places output on the standard output stream *stdout*. *sprintf* places "output," followed by the null character (**\0**), in consecutive bytes starting at **s*; it is the user's responsibility to ensure that enough storage is available. Each function returns the number of characters transmitted (not including the **\0** in the case of *sprintf*), or a negative value if an output error was encountered.

Each of these functions converts, formats, and prints its *args* under control of the *format*. The *format* is a character string that contains two types of objects: plain characters, which are simply copied to the output stream, and conversion specifications, each of which results in fetching of zero or more *args*. The results are undefined if there are insufficient *args* for the format. If the format is exhausted while *args* remain, the excess *args* are simply ignored.

Each conversion specification is introduced by the character %. After the %, the following appear in sequence:

Zero or more *flags*, which modify the meaning of the conversion specification.

An optional decimal digit string specifying a minimum *field width*. If the converted value has fewer characters than the field width, it will be padded on the left (or right, if the left-adjustment flag '—', described below, has been given) to the field width. If the field width for an **s** conversion is preceded by a 0, the string is right adjusted with zero-padding on the left.

A *precision* that gives the minimum number of digits to appear for the **d**, **o**, **u**, **x**, or **X** conversions, or the maximum number of characters to be printed from a string in **s** conversion. The precision takes the form of a period (.) followed by a decimal digit string; a null digit string is treated as zero.

An optional **l** (ell) specifying that a following **d**, **o**, **u**, **x**, or **X** conversion character applies to a long integer *arg*. An **l** before any other conversion character is ignored.

A character that indicates the type of conversion to be applied.

A field width of precision may be indicated by an asterisk (*) instead of a digit string. In this case, an integer *arg* supplied the field width or precision. The *arg* that is actually converted is not fetched until the conversion letter is seen, so the *args* specifying field width or precision must appear *before* the *arg* (if any) to be converted.

The flag characters and their meanings are:

–	The result of the conversion will be left-justified within the field.
+	The result of a signed conversion will always begin with a sign (+ or –).
blank	If the first character of a signed conversion is not a sign, a blank will be prefixed to the result. This implies that if the blank and + flags both appear, the blank flag will be ignored.
#	This flag specifies that the value is to be converted to an "alternate form." For **c**, **d**, **s**, and **u** conversions, the flag has no effect. For **o** conversion, it increases the precision to force the first digit of the result to be a zero. For **x** or **X** conversion, a non-zero result will have **0x** or **0X** prefixed to it.

The conversion characters and their meanings are:

d,o,u,x,x	The integer *arg* is converted to signed decimal, unsigned octal, decimal, or hexadecimal notation (**x** and **X**) respectively; the letters **abcdef** are used for **x** conversion and the letters **ABCDEF** for **X** conversion. The precision specifies the minimum number of digits to appear; if the value being converted can be represented in fewer digits, it will be explained with leading zeros. (For compatibility with older versions, padding with leading zeros may alternatively be specified by prepending a zero to the field width. This does not imply an actual value for the field width.) The default precision is 1. The result of converting a zero value with a precision of zero is a null string.
c	The character *arg* is printed.
s	The *arg* is taken to be a string (character pointer) and characters from the string are printed until a null character (**\0**) is encountered or the number of characters indicated by the precision specification is reached. If the precision is missing, it is taken to be infinite so all characters up to the first null character are printed. A NULL value for *arg* will yield undefined results.
%	Print a %; no argument is converted.

In no case does a nonexistent or small field width cause truncation of a field; if the result of a conversion is wider than the field width, the field is simply expanded to contain the conversion result. Characters generated by *printf* are printed as if *putc(3S)* had been called.

EXAMPLE

To print a data and time in the form "Sunday, July 3, 10:02," where *weekday* and *month* are pointers to null-terminated strings:

```
printf("%s, %s %d, %d:%.2d", weekday, month, day, hour, min);
```

SEE ALSO

ecvt(3C), putc(3S), scanf, stdio(3S)

NAME

rand, srand — simple random-number generator

SYNOPSIS

int rand ( )

void srand (seed)
unsigned seed;

DESCRIPTION

rand uses a multiplicative congruential random-number generator with period 2^{32} that returns successive pseudo-random numbers in the range from 0 to $2^{15}-1$.

srand can be called at any time to reset the random-number generator to a random starting point. The generator is initially seeded with a value of 1.

NOTE

The spectral properties of *rand* leave a great deal to be desired. *drand48(3C)* provides a much better, though more elaborate, random-number generator.

SEE ALSO

drand48(3C)

NAME

scanf, sscanf — convert formatted input

SYNOPSIS

#include <stdio.h>

int scanf (format [, pointer] . . .)
char *format;

int sscanf (s, format [, pointer] . . .)
char *s, *format;

DESCRIPTION

scanf reads from the standard input stream *stdin*. *sscanf* reads from the character string *s*. Each function reads characters, interprets them according to a format, and stores the results in its arguments. Each expects, as arguments, a control string *format* described below, and a set of *pointer* arguments indicating where the converted input should be stored.

The control string usually contains conversion specifications, which are used to direct interpretation of input sequences. The control string may contain:

1. White-space characters (blanks, tabs, new-lines, or form-feeds) which, except in two cases described below, cause input to be read up to the next non-white-space character.
2. An ordinary character (not %), which must match the next character of the input stream.
3. Conversion specifications, consisting of the character %, an optional assignment suppressing character *, an optional numerical maximum field width, an optional **l** (ell) or **h** indicating the size of the receiving variable, and a conversion code.

A conversion specification directs the conversion of the next input field; the result is placed in the variable pointed to by the corresponding argument, unless assignment suppression was indicated by *. The suppression of assignment provides a way of describing an input field which is to be skipped. An input field is defined as a string of non-space characters; it extends to the next inappropriate character or until the field width, if specified, is exhausted. For all descriptors except "[" and "c", white space leading an input field is ignored.

The conversion code indicates the interpretation of the input field; the corresponding pointer argument must usually be of a resticted type. For a suppressed field, no pointer argument is given. The following conversion codes are legal:

% a single % is expected in the input at this point; no assignment is done.

d a decimal integer is expected; the corresponding argument should be an integer pointer.

u an unsigned decimal integer is expected; the corresponding argument should be an unsigned integer pointer.

o an octal integer is expected; the corresponding argument should be an integer pointer.

x a hexadecimal integer is expected; the corresponding argument should be an integer pointer.

s a character string is expected; the corresponding argument should be a character pointer pointing to an array of characters large enough to accept the string and a terminating **\0**, which will be added automatically. The input field is terminated by a white-space character.

c a character is expected; the corresponding argument should be a character pointer. The normal skip over white space is suppressed in this case; to read the next non-space character, use % **1s**. If a field width is given, the corresponding argument should refer to a character array; the indicated number of characters is read.

[indicates string data and the normal skip over leading white space is suppressed. The left bracket is followed by a set of characters, which we will call the *scanset*, and a right bracket; the input field is the maximal sequence of input characters consisting entirely of characters in the scanset. The circumflex (^), when it appears as the first character in the scanset, serves as a complement operator and redefines the scanset as the set of all characters *not* contained in the remainder of the scanset string. There are some conventions used in the construction of the scanset. A range of characters may be represented by the construct *first–last*, thus [0123456789] may be expressed [0—9]. Using this convention, *first* must be lexically less than or equal to *last*, or else the dash will stand for itself. The dash will also stand for itself whenever it is the first or the last character in the scanset. To include the right square bracket as an element of the scanset, it must appear as the first character (possibly preceded by a circumflex) of the scanset, and in this case it will ot be syntactically interpreted as the closing bracket. The corresponding argument must point to a character array large enough to hold the data field and the terminating **\0**, which will be added automatically. At least one character must match for this conversion to be considered successful.

The conversion characters **d**, **u**, **o**, and **x** may be preceded by **l** or **h** to indicate that a pointer to **long** or to **short** rather than to **int** is in the argument list. The **l** or **h** modifier is ignored for other conversion characters.

scanf conversion terminates at **EOF**, at the end of the control string, or when an input character conflicts with the control string. In the latter case, the offending character is left unread in the input steam.

scanf returns the number of successfully matched and assigned input items; this number can be zero in the event of an early conflict between an input character and the control string. If the input ends before the first conflict or conversion, **EOF** is returned.

EXAMPLES

The call:

```
int i, n; float x; char name[50];
n = scanf ("%d%f%s", &i, &x, name);
```

with the input line:

```
25 54.32E−1 thompson
```

will assign to *n* the value **3**, to *i* the value **25**, to *x* the value **5.432**, and *name* will contain **thompson\0**. Or:

```
int i; float x; char name[50];
(void) scanf("%2d%f%*d %[0−9]", &i, &x, name);
```

with input:

```
56789 0123 56a72
```

will assign **56** to *i*, **789.0** to *x*, skip **0123**, and place the string 56\0 in *name*. The next call to *getchar* (see *getc*(3S)) will return **a**.

SEE ALSO

getc(3S), printf, strtod(3C), strtol

NOTE

Trailing white space (including a new-line) is left unread unless matched in the control string.

DIAGNOSTICS

These functions return **EOF** on end of input and a short count for missing or illegal data items.

BUGS

The success of literal matches and suppressed assignments is not directly determinable.

NAME

sputl, sgetl — access long integer data in a machine independent fashion.

SYNOPSIS

```
void sputl (value, buffer)
long value;
char *buffer;

long sgetl (buffer)
char *buffer;
```

DESCRIPTION

sputl takes the four bytes of the long integer *value* and places them in memory starting at the address pointed to by *buffer*. The ordering of the bytes is the same across all machines.

sgetl retrieves the four bytes in memory starting at the address pointed to by *buffer* and returns the long integer value in the byte ordering of the host machine.

The combination of *sputl* and *sgetl* provides a machine-independent way of storing long numeric data in a file in binary form without conversion to characters.

A program which uses these functions must be loaded with the object-file access routine library **LIBDIR/libld.a**.

NAME

strcat, strncat, strcmp, strncmp, strcpy, strncpy, strlen, strchr, strrchr, strpbrk, strspn, strcspn, strtok — string operations.

SYNOPSIS

```
#include <string.h>

char *strcat (s1, s2)
char *s1, *s2;

char *strncat (s1, s2, n)
char *s1, *s2;
int n;

int strcmp (s1, s2)
char *s1, *s2;

int strncmp (s1, s2, n)
char *s1, *s2;
int n;

char *strcpy (s1, s2)
char *s1, *s2;

char *strncpy (s1, s2, n)
char *s1, *s2;
int n;

int strlen (s)
char *s;

char *strchr (s, c)
char *s;
int c;

char *strrchr (s, c)
char *s;
int c;

char *strpbrk (s1, s2)
char *s1, *s2;

int strspn (s1, s2)
char *s1, *s2;
```

int strcspn (s1, s2)
char *s1, *s2;

char *strtok (s1, s2)
char *s1, *s2;

DESCRIPTION

The arguments **s1, s2** and **s** point to strings (arrays of characters terminated by a null character). The functions *stract*, *strncat*, *strcpy*, and *strncpy* all alter **s1**. These functions do not check for overflow of the array pointed to by **s1**.

strcat appends a copy of string **s2** to the end of string **s1**. *strncat* appends at most **n** characters. Each returns a pointer to the null-terminated result.

strcmp compares its arguments and returns an integer less than, equal to, or greater than 0, according as **s1** is lexicographically less than, equal to, or greater than **s2**. *strncmp* makes the same comparison but looks at most **n** characters.

strcpy copies string **s2** to **s1**, stopping after the null character has been copied. *strncpy* copies exactly **n** characters, truncating **s2** or adding null characters to **s1** if necessary. The result will not be null-terminated if the length of **s2** is **n** or more. Each function returns **s1**.

strlen returns the number of characters in **s**, not including the terminating null character.

strchr (*strrchr*) returns a pointer to the first (last) occurrence of character **c** in string **s**, or a NULL pointer if **c** does not occur in the string. The null character terminating a string is considered to be part of the string.

strpbrk returns a pointer to the first occurrence in string **s1** of any character from string **s2**, or a NULL pointer if no character from **s2** exists in **s1**.

strspn (*strcspn*) returns the length of the initial segment of string **s1** which consists entirely of characters from (not from) string **s2**.

strtok considers the string **s1** to consist of a sequence of zero or more text tokens separated by spans or one more characters from the separator string **s2**. The first call (with pointer **s1** specified) returns a pointer to the first character of the first token, and will have written a null character into **s1** immediately following the returned token. The function keeps track of its position in the string between separate calls, so that subsequent calls (which must be made with the first argument a NULL pointer) will work through the string **s1** immediately following that token. In this way, subsequent calls will work through the string **s1** until no tokens remain. The separator string **s2** may be different from call to call. When no token remains in **s1**, a NULL pointer is returned.

NOTE

For user convenience, all these functions are declared in the optional *<string.h>* header file.

BUGS

strcmp and *strncmp* use native character comparison, which is signed on *PDP* 11 Computers and *VAX* 11 Computers, unsigned on other machines. Thus the sign of the value returned when one of the characters has its high-order bit set is implementation-dependent.

Character movement is performed differently in different implementations. Thus overlapping moves may yield surprises.

NAME

strtol, atol, atoi — convert string to integer

SYNOPSIS

long strtol (str, ptr, base)
char *str, **ptr;
int base;

long atol (str)
char *str;

int atoi (str)
char *str;

DESCRIPTION

strtol returns as a long integer the value represented by the character string pointed to by *str*. The string is scanned up to the first character inconsistent with the base. Leading "white-space" characters (as defined by *isspace* in *ctype*) are ignored.

If the value of *ptr* is not (char **)NULL, a pointer to the character terminating the scan is returned in the location pointed to by *ptr*. If no integer can be formed, that location is set to *str*, and zero is returned.

If *base* is positive (and not greater than 36), it is used as the base for conversion. After an optional leading sign, leading zeros are ignored, and "0x" or "0X" is ignored if *base* is 16.

If *base* is zero, the string itself determines the base thusly: After an optional leading sign a leading zero indicates octal conversion, and a leading "0x" or "0X" hexadecimal conversion. Otherwise, decimal conversion is used.

Truncation from *long* to *int* can, of course, take place upon assignment or by an explicit cast.

atol(str) is equivalent to *strtol(str, (char **)NULL, 10).*
atoi(str) is equivalent to *(int) strtol(str, (char**)NULL, 10).*

SEE ALSO

ctype, scanf, strtod(3)

BUGS

Overflow conditions are ignored.

NAME

swab — swap bytes

SYNOPSIS

void swab (from, to, nbytes)
char *from, *to;
int nybtes;

DESCRIPTION

swab copies *nbytes* bytes pointed to by *from* to the array pointed to by *to*, exchanging adjacent even and odd bytes. It is useful for carrying binary data between *PDP* 11 Computers and other machines. *nbytes* should be even and nonnegative. If *nbytes* is odd and positive *swab* uses nbytes−1 instead. If *nbytes* is negative, *swab* does nothing.

NAME

intro — Introduction to *WE* 32100 Microprocessor File Formats and Include Files

DESCRIPTION

This section describes the header files and file formats used. C **struct** declarations appear where useful. Several of the files apply to the object file format. The others are useful for assembly language programming or for installation of the various pieces of the processor. Normally, these files reside in directories under **/usr/m32**. Specific installations may alter this directory as described in *paths*.

Briefly, three main directories contain any files for users. All descriptions of these files use the names **BINDIR**, **INCDIR**, and **LIBDIR** for the command, include, and library directories, respectively. They are set at build time.

SEE ALSO

paths

NAME

filehdr — File Header for *WE* 32100 Microprocessor Object File

SYNOPSIS

#include "filehdr.h"

DESCRIPTION

Every object file begins with a 20-byte header. The following C **struct** declaration is used:

```
struct   filehdr
{
        unsigned short   f_magic ;    /* magic number */
        unsigned short   f_nscns ;    /* number of sections */
        long             f_timdat ;   /* time & date stamp */
        long             f_symptr ;   /* file ptr to symtab */
        long             f_nsyms ;    /* # symtab entries */
        unsigned short   f_opthdr ;   /* sizeof(opt hdr) */
        unsigned short   f_flags ;    /* flags */
};
```

f_symptr is the byte offset into the file at which the symbol table can be found. Its value can be used as the offset in the *UNIX* System *fseek* command to position an I/O stream to the symbol table. The processor uses the optional header for a *UNIX* System header, which is always 28 bytes. The only valid processor magic number is:

```
#define FBOMAGIC  0560
#define RBOMAGIC  0562
```

The value in *f_timdat* is obtained from the *time* system call. Flag bits currently defined are:

```
#define F_RELFLG     00001/* relocation entries stripped */
#define F_EXEC       00002/* file is executable */
#define F_LNNO       00004/* line numbers stripped */
#define F_LSYMS      00010/* local symbols stripped */
#define F_MINMAL     00020/* minimal object file */
#define F_UPDATE     00040/* update file, ogen produced */
#define F_SWABD      00100/* file is "pre-swabbed" */
#define F_AR16WR     00200/* 16 bit DEC host */
#define F_AR32WR     00400/* 32 bit DEC host */
#define F_AR32W      01000/* non-DEC host */
#define F_PATCH      02000/* "patch"list in opt hdr */
#define F_BM32B      02000/*file contains WE 32100 code */
```

SEE ALSO

m32a.out

NAME

ldfcn — *WE* 32100 Microprocessor Object File Access Routines

SYNOPSIS

#include <stdio.h>
#include "INCDIR/filehdr.h"
#include "INCDIR/ldfcn.h"

DESCRIPTION

The object file access routines are a collection of functions for reading an object file that is in common object file form. Although the calling program must know the detailed structure of the parts of the object file that it processes, the routines effectively insulate the calling program from knowledge of the overall structure of the object file.

The interface between the calling program and the object file access routines is based on the defined type **LDFILE**, defined as **struct ldfile**, declared in the header file **ldfcn.h**. The primary purpose of this structure is to provide uniform access to both simple object files and to object files that are members of an archive file.

The function *ldopen* allocates and initializes the **LDFILE** structure and returns a pointer to the structure to the calling program. The fields of the **LDFILE** structure may be accessed individually through macros defined in **ldfcn.h** and contain the following information:

LDFILE *ldptr;

TYPE(ldptr)	The file magic number, used to distinguish between archive members and simple object files.
IOPTR(ldptr)	The file pointer returned by *fopen* and used by the standard input/output functions.
OFFSET(ldptr)	The file address of the beginning of the object file; the offset is non-zero if the object file is a member of an archive file.
HEADER(ldptr)	The file header structure of the object file.

The object file access functions themselves may be divided into four categories:

(1) Functions that open or close an object file:
ldopen and *ldaopen* open a processor object file,
ldclose and *ldaclose* close a processor object file.

(2) Functions that read header or symbol table information:
ldahread reads the archive header of a member of an archive file,
ldfhread reads the file header of an object file,
ldshread and *ldnshread* read a section header of an object file,
ldtbread reads a symbol table entry of an object file.
ldgetname retrieves a symbol name from a symbol table entry or from the string table.

(3) Functions that position an object file at (seek to) the start of the section, relocation, or line number information for a particular section:

ldohseek seeks to the optional file header of an object file,
ldsseek and *ldnsseek* seek to a section of an object file,
ldrseek and *ldnrseek* seek to the relation information for a section of an object file,
ldlseek and *ldnlseek* seek to the line number information for a section of an object file,
ldtbseek seek to the symbol table of an object file.

(4) The function *ldtbindex* which returns the index of a particular object file symbol table entry.

These functions are described in detail on their respective manual pages.

All the functions except *ldopen, ldaopen, ldgetname*, and *ldtbindex* return either **SUCCESS** or **FAILURE**, both constants defined in **ldfcn.h**. *ldopen* and *ldaopen* both return pointers to a **LDFILE** structure.

MACROS

Additional access to an object file is provided through a set of macros defined in *ldfcn.h*. These macros parallel the standard input/output file reading and manipulating functions, translating a reference of the **LDFILE** structure into a reference to its file descriptor field.

The following macros are provided:

LDFILE *ldptr;
GETC(ldptr)
FGETC(lpdptr)
GETW(ldptr)
UNGETC(c,ldptr)
FGETS(s, n, ldptr)
FRED((char *)ptr, sizeof (*ptr), nitems, ldptr)
FSEEK(ldptr, offset, ptrname)
FTELL(ldptr)
REWIND(ldptr)
FEOF(ldptr)
FERROR(ldptr)
FILENO(ldptr)
SETBUF(ldptr, buf)
STROFFSET(ldptr)

The STROFFSET macro calculates the address of the string table in an object file.

See the manual pages for the corresponding standard input/output library functions for details on the use of the rest of these macros.

The program must be loaded with the object file access routine library **LIBDIR/libld.a**.

intro describes *LIBDIR* and *INCDIR*.

CAVEATS

The macro **FSEEK** defined in the header file **ldfcn.h** translates into a call to the standard input/output *UNIX* System function *fseek*. **FSEEK** should not be used to seek from the end of an archive file since the end of an archive file may not be the same as the end of one of its object file members!

SEE ALSO

ldahread, ldclose, ldfhread, ldlread, ldlseek, ldohseek, ldopen, ldrseek, ldlseek, ldshread, ldtbindex, ldtbread, ldtbseek, intro, ldfcn, paths

NAME

linenum — Line Number Entries in a *WE* 32100 Microprocessor Object File

SYNOPSIS

#include "linenum.h"

DESCRIPTION

Compilers based on *pcc* generate an entry in the object file for each C source line on which a breakpoint is possible. Users can then reference line numbers when using the appropriate software test system. The structure of these line number entries appears below.

```
struct    lineno
{
          union
          {
                    long      l_symndx ;
                    long      l_paddr ;
          }                   l_addr ;
          Unsigned  short     l_lnno ;
};
```

Numbering starts with one for each function. The initial line number entry for a function has *l_lnno* equal to zero, and the symbol table index of the function's entry is in *l_symndx*. Otherwise, *l_lnno* is non-zero and *l_paddr* is the physical address of the code for the referenced line. Thus the overall structure is the following:

l_addr	*l_lnno*
function symtab index	0
physical address	line
physical address	line
...	
function symtab index	0
physical address	line
physical address	line
...	

SEE ALSO

m32cc, m32a.out.

NAME

m32a.out — *WE* 32100 Microprocessor Object File Format

DESCRIPTION

This describes the default output file format from the **m32as** assembler, and the **m32ld** link editor. The resultant file can be executed on the target machine if no errors or unresolved references were found. In no case is the file given *UNIX* System execute permissions because the object code is for the target machine; not the host machine on which the file was created.

An object file supports user-defined sections and contains extensive information for symbolic software testing. The overall structure of an object file is given below.

File header.
UNIX System header.
Section 1 header.
...
Section n header.
Section 1 data.
...
Section n data.
Section 1 relocation.
...
Section n relocation.
Section 1 line numbers.
...
Section n line numbers.
Symbol table.

See *filehdr, scnhdr, reloc, linenum,* and *syms* for descriptions of the individual parts. Every section created by **m32as** contains a multiple of four number of bytes; directives to **m32ld** can create a section with an odd number of bytes.

A set of functions exist to manipulate object files. See *ldfcn* and its associated references for more information.

SEE ALSO

m32as, m32ld, ldfcn, filehdr, linenum, reloc, schhdr, syms.

NAME

paths — Directory Path Names for the *WE* 32100 Microprocessor

SYNOPSIS

#include "paths.h"

DESCRIPTION

Users may install the SGP under four separate directories: **bin** for the commands, **lib** for the libraries, **include** for the header files, and **man** for manual pages. After the SGP is installed, the directories should not be moved. Users must specify the installation directories by using the "pathedit" command before installation. "Pathedit" is described in the README file delivered with the SGP. The following are among the modified definitions:

```
#define   BINDIR    "/usr/m32/bin"
#define   INCDIR    "/usr/m32/include"
#define   LIBDIR    "/usr/m32/lib"
#define   MANDIR    "/usr/m32/man"
```

Additionally, users may specify a directory that processor tools should use for temporary files.

```
#define   TMPDIR    "/usr/tmp"
```

NAME

reloc — Relocation Information for a *WE* 32100 Microprocessor Object File

SYNOPSIS

#include "reloc.h"

DESCRIPTION

Object files have one relocation entry for each relocatable reference in the text or data. If relocation information is present, it is in the following format:

```
struct    reloc
{
          long        r_vaddr ;    /* (virtual) address of reference */
          long        r_symndx ;   /* index into symbol table */
          short       r_type ;     /* relocation type */
};
#define   R_ABS       0
#define   R_DIR32     06
#define   R_DIR32S    012
```

As the link editor reads each input section and performs relocation, the relocation entries are read. They direct how references found within the input section are treated.

R_ABS — The reference is absolute, and no relocation is necessary. The entry is ignored.

R_DIR32 — A direct, 32-bit reference to a symbol's virtual address.

R_DIR32S — A direct, 32-bit reference to a symbol's virtual address. The 32-bit value is stored in reverse order in the object file.

Other relocation types will be defined as they are needed.

Relocation entries are generated automatically by the compiler and the assembler, and automatically utilized by the link editor. A link editor option exists for removing the relocation entries from an object file.

SEE ALSO

m32ld, m32strip, m32a.out, syms.

NAME

scnhdr — Section Header for a *WE* 32100 Microprocessor Object File

SYNOPSIS

#include "scnhdr.h"

DESCRIPTION

Every object file has a table of section headers to specify the layout of the data within the file. Each section within an object file has its own header. The C structure appears below.

```
struct  scnhdr
{
        char            s_name[8] ;
        long            s_paddr ;       /* physical address */
        long            s_vaddr ;       /* virtual address */
        long            s_size ;        /* section size */
        long            s_scnptr ;      /* file ptr to raw data */
        long            s_relptr ;      /* file ptr to relocation */
        long            s_lnnoptr ;     /* file ptr to line numbers */
        unsigned short  s_nreloc ;      /* # reloc entries */
        unsigned short  s_nlnno ;       /* # line number entries */
        long            s_flag ;        /* flags */
};
```

File pointers are byte offsets into the file; they can be used as the offset in a call to the *UNIX* System command *fseek*. If a section is initialized, the file contains the actual bytes. An uninitialized section is somewhat different. It has a size, symbols defined in it, and symbols that refer to it. But it can have no relocation entries, line numbers, or data. Consequently, an uninitialized section has no raw data in the object file, and the values for *s_scnptr, s_relptr, s_lnnoptr, s_nreloc,* and *s_nlnno* are zero.

SEE ALSO

m32ld, m32a.out.

NAME

syms — *WE* 32100 Microprocessor Object File Symbol Table Format

SYNOPSIS

#include "syms.h"

DESCRIPTION

Processor object files contain information to support *symbolic* software testing. Line number entries, *linenum*, and extensive symbolic information permit testing at the C *source* level. The symbol table for every object file is organized as:

```
File name 1.
        Function 1.
                Local symbols for function 1.
        Function 2.
                Local symbols for function 2.
        ...
        Static externs for file 1.

File name 2.
        Function 1.
                Local symbols for function 1.
        Function 2.
                Local symbols for function 2.
        ...
        Static externs for file 2.
...

Defined global symbols.
Undefined global symbols.
```

The entry for a symbol is a fixed-length structure. The members of the structure hold the name (null padded), its value, and other information. The C structure is:

```
#define  SYMNMLEN 8
#define  FILNMLEN 14

struct   syment
{
union
{
        char _n_name[SYMNMLEN];  /* symbol name */
        struct
        {
                                        long _n_zeros;   /* ==OL when in
                                                            string table */
                                        long _n_offset;  /* location of
                                                            name in table */
```

```
        } _n_n;
        char *_n_nptr[2];           /* allows overlaying */
} _n;

        unsigned long               n_value ;/* value of symbol */
        short                       n_scnum ;/* section number */
        unsigned short              n_type ;/* type and derived type */
        char                        n_sclass ;/* storage class */
        char                        n_numaux ;/* number of aux entries */
};

#define n_name _n._n_name
#define n_zeros _n._n_n._n_zeros
#define n_offset _n._n_n._n_offset
#define n_nptr _n._n_nptr[1]
```

Some symbols require more information than a single entry; they are followed by *auxiliary entries* that are the same size as a symbol entry. The format is:

```
union auxent
{
    struct
    {
            long                x_tagndx;
            union
            {
                    struct
                    {
                            unsigned short x_lnno;
                            unsigned short x_size;
                    }x_lnsz;
                    long        x_fsize;
            }x_misc;
            union
            {
                    struct
                    {
                            long x_lnnoptr;
                            long x_endndx;
                    } x_fcn;
                    struct
                    {
                            unsigned short x_dimen[DIMNUM];
                    }           x-ary;
            }                   x_fcnary;
            unsigned short      x_tvndx;
```

```
}           x_sym;
struct
{

            char        x_fname[FILNMLEN];
}           x_file;
 struct
 {
                        long x_scnlen;
                        unsigned short   x_nreloc:
                        unsigned short   x_nlinno:
 }          x_scn;
struct
{
                        long x_tvfill;
                        unsigned short   x_tvlen;
                        unsigned short   x_tvran[2];
            }           x_tv;
};
```

Indices of symbol table entries begin at *zero*.

SEE ALSO

m32a.out, *linenum*.

Glossary and Acronyms

Absolute deferred mode - An address mode that uses an address embedded in the operand to locate a pointer to data.

Absolute mode - An address mode that uses an address embedded in the operand to locate data.

Address arithmetic unit (AAU) - Fetch unit element that calculates 32-bit addresses.

Argument pointer (AP) - User register that points to the beginning location in the stack where a set of arguments for a function has been pushed.

Arithmetic logic unit (ALU) - On-chip unit that performs arithmetic operations on 32-bit data.

Assert - To drive a signal to its active state.

Bit field - A sequence of 1 to 32 bits contained in a base word. The field is specified by the address of its base word, a bit offset, and a width.

Bit offset - Identifies the starting bit of the field in its base word. The offset ranges from 0 to 31.

Bus interface control - Provides all the strobes and control signals necessary to implement the interface with peripherals.

Byte - An 8-bit quantity that may appear at any address in memory.

Cache - A high-speed memory filled at a lower speed from main memory. Used to reduce memory access time.

Cache disable (CD) - A field in the PSW that enables and disables the instruction cache.

Cache flush disable (CFD) - A field in the processor status word (PSW) that enables and disables instruction cache flushing (emptying of the cache's contents) when a new process is loaded via the XSWITCH_TWO microsequence.

Condition codes (NZVC) - The flags in this 4-bit field reflect the resulting status of the most recent instruction execution which affects them. The four flags are negative (N), zero (Z), overflow (V), and carry (C).

Coprocessor - A support processor that operates synchronously with the CPU to provide greater throughput in arithmetic or I/O functions.

Current execution level (CM) - A 2-bit field in the PSW that represents the current execution level. The four execution levels are kernel, executive, supervisor, and user.

Dedicated registers - Seven registers (r9—r15) that have specific, predetermined functions.

Displacement mode - An address mode that uses a register and an offset, both embedded in the operand, added together to form the address of data.

Displacement deferred mode - An address mode that uses a register and an offset, both embedded in the operand, added together to form the address of a pointer to data.

Enable overflow trap (OE) - A field in the PSW that enables overflow traps.

Exception type (ET) - A 2-bit field in the PSW that indicates exceptions generated during operations. The four types of exceptions are normal, stack, process, and reset.

Exceptional conditions - Events other than interrupts and reset requests that may interrupt the execution of a program. The four classes of exceptional conditions are normal exceptions, stack exceptions, process exceptions, and reset exceptions.

Execute unit - The elements in this unit perform all arithmetic and logic operations, perform all shift and all rotate operations, and compute the condition flags.

Expanded-operand-type mode - An address mode that changes the type of the instruction for an operand and those that follow it in the instruction. It does not affect immediate operands.

Faults - Error conditions that are detected outside the microprocessor and conveyed to the microprocessor over its fault input $\overline{\text{FAULT}}$.

Fetch unit - The elements in this unit handle the instruction stream and perform memory-based operand accesses.

Frame pointer (FP) - User register that points to the beginning location in the stack of a function's local variables.

Full interrupt - Interrupt whose handling routine implements a process switch to the interrupt's handler. All interrupts are handled via the full interrupt sequence if the QIE bit in the PSW is cleared (0).

General-purpose registers - Nine registers (r0—r8) that may be used for high-speed accumulation, for addressing, or for temporary data storage.

Halfword - 16-bit quantity that may appear at any address in memory that is divisible by 2.

Immediate and displacement extractor - Provides address calculation data to the address arithmetic unit (AAU) for its use in calculating 32-bit addresses.

Immediate mode - An address mode where the operand contains actual data to be used by instruction.

Instruction cache - A 64- by 32-bit on-chip cache used to increase the microprocessor's performance by reducing external memory reads for instruction fetches.

Instruction queue - An 8-byte, first-in-first-out (FIFO) on-chip queue that stores prefetched instructions.

Internal state code (ISC) - A 4-bit field in the PSW that distinguishes between exceptions of the same exception type.

Interrupt - A means by which external devices may request service by the microprocessor.

Interrupt priority level (IPL) - A 4-bit field in the PSW that represents the current interrupt priority level.

Interrupt stack pointer (ISP) - User register that contains the 32-bit memory address of the top of the interrupt stack.

Main controller - The microprocessor's central control unit. It is responsible for acquiring and decoding instruction opcodes and directing the action of the fetch and execute controllers.

Memory management unit (MMU) - A software or hardware unit, or combination of both, that translates virtual addresses into physical addresses and verifies access authorization. The *WE* 32101 Memory Management Unit provides this function for the CPU.

Negate - To drive a signal to its inactive state.

Nonmaskable interrupt - Type of interrupt that interrupts the microprocessor regardless of the priority level in the IPL field of the PSW.

Normal exceptions - A class of exceptional conditions generated by the microprocessor when it detects a condition such as a trap, invalid opcode, or illegal operation.

Operand descriptor - First byte of an operand defining which address mode and register the operand uses.

Pipelining - Overlapping the execution of instructions to increase the microprocessor's performance.

Prefetch - A technique where the CPU fetches an instruction prior to the completion of previous instructions.

Previous execution level (PM) - A 2-bit field in the PSW that represents the previous execution level. The four execution levels are kernel, executive, supervisor, and user.

Privileged instruction - An operating system group instruction that can execute only in kernel execution level.

Process control block (PCB) - A process data structure in external memory that saves the context of a process when the process is not running. This context consists of the initial and current contents of control registers (PSW, PC, and SP), the last contents of registers r0 through r10, boundaries for an execution stack, and memory specifications for the process.

Process control block pointer (PCBP) - User register that points to the starting address of the process control block for the current process.

Process exceptions - A class of exceptional conditions that may occur during a process switch.

Processor status word (PSW) - User register that contains status information about the microprocessor and the current process.

Program counter (PC) - User register that contains the 32-bit memory address of the instruction being executed or, upon completion, contains the starting address of the next instruction to be executed.

Quick interrupt - An interrupt whose handling routine pushes the old PSW and PC on the stack and fetches a new PSW and PC that correspond to the interrupt's handler. For this reason the quick interrupt handling routine requires less time than a full interrupt which implements a process switch to the interrupt's handler. Interrupts are handled via the quick-interrupt sequence if the QIE bit in the PSW is set (1).

Quick-interrupt enable (QIE) - A field in the PSW that enables and disables the quick-interrupt facility.

Read interlocked operation - An operation which consists of a memory fetch (read access), one or more internal microprocessor operations, and then a write access to the same memory location.

Register deferred mode - An address mode that uses a register name, embedded in an operand, which contains a pointer to data to be used by the instruction.

Register mode - An address mode that uses a register name, embedded in an operand, which contains data to be used by the instruction.

Register-initial context (RI) - A 2-bit field in the PSW that controls the microprocessor context switching strategy.

Reset exceptions - A class of exceptional conditions that is triggered by an error condition in accessing critical system data.

Scratch registers - User registers r0, r1, and r2. These three registers are used by the C compiler to store temporary values and also return specific values during procedure calls.

Short offset mode - An address mode that uses an offset embedded in an operand. The offset is added to the frame pointer or argument pointer to form the address of data.

Sign extension - Automatic extension of a byte or halfword value to 32 bits by filling the high-order bits with the value of the sign bit.

Stack exceptions - A class of exceptional conditions that may occur during a process switch or a GATE sequence.

Stack pointer (SP) - User register that contains the current 32-bit address of the top of the execution stack; i.e., the memory address of the next item to be stored on (pushed onto) the stack or the last item retrieved (popped) from the stack.

Trace enable (TE) - A field in the PSW that enables the trace function.

Trace mask (TM) - A field in the PSW that enables masking of a trace trap.

Trace mechanism - An interpretive diagnostic trace trap using two bits in the PSW, trace enable (TE) and trace mask (TM), to analyze each executed instruction.

User registers - The sixteen 32-bit registers (r0—r15) that are available to the user. The user registers consist of nine general purpose registers (r0—r8) and seven dedicated registers (r9—r15).

Vestigial cycle - A clock cycle that follows every access and is provided to allow enough time for a memory management unit to release the shared address bus.

Wait-state - Idle periods that may be generated during a bus transaction to allow slow peripherals to handshake with the microprocessor.

Width - The size of a bit field. Width plus one is the number of bits in the field. The width ranges from 0 to 31.

Word - A 32-bit quantity that may appear at any address divisible by 4.

Working registers - Registers that are used exclusively by the microprocessor and are not user-accessible.

Zero extension - Automatically extending a byte or halfword value to 32 bits by filling the high-order bits with zeros.

3-state - To place an input in a high-impedance state.

AAU - Address arithmetic unit
ALU - Arithmetic logic unit
AP - Argument pointer
BPT - Breakpoint trap
BSS - Bounded static storage
C - Condition flag bit carry
CALLPS - Call process
CD - Cache disable
CFD - Cache flush disable
CM - Current execution level
CMOS - Complimentary metal-oxide semiconductor
COFF - Common object file format
COPY - "Copy" section
CPU - Central processing unit
CR - Configuration register
DMA - Direct memory access
DSECT - "Dummy" section
EPROM - Eraseable programmable ROM
ET - Exception type
FLTAR - Fault address register
FLTCR - Fault code register
FP - Frame pointer
I/O - Input/output
IPL - Interrupt priority level
ISC - Internal state code
ISP - Interrupt stack pointer
LIFO - Last-in-first-out
LRU - Least recently used
LSB - Least significant bit
mmmm - Mode field
MMU - Memory management unit
MSB - Most significant bit
N - Condition flag bit negative
NOP - No operation
OE - Overflow enable
PC - Program counter
PCB - Process control block
PCBP - Process control block pointer
PD - Page descriptors
PDT - Page descriptor table
PM - Previous execution level
POT - Page offset field
PPC - Prefetch counter
PSL - Page select field
PSW - Processor status word
QIE - Quick-interrupt enable
RAM - Random access read/write memory
RI - Register-initial
ROM - Read-only memory
rrrr - Register field
RSB - Return from subroutine
SD - Segment descriptors
SDP - Software demand paging
SDT - Segment descriptor table
SGP - Software generation programs
SID - Section ID field
SOT - Segment offset field
SP - Stack pointer
SSL - Segment select field
TE - Trace enable
TM - Trace mask
TT - Trace trap
TTL - Transistor-transistor logic
V - Condition flag bit overflow
Z - Condition flag bit zero

Index

A

B

C

D

E

F

G

M

P

Q

R

S

T

U

V

W

X

Z