AT&T

305-000
March 1986

AT&T 3B2/3B5/3B15 Computers
Assembly Language Programming Manual

ACKNOWLEDGEMENTS

Prepared and published by
Document Development Organization — Microelectronics Projects Group
AT&T Network Systems, Morristown

for the

Microsystems Product Management
AT&T Technologies, Inc.

and the

AT&T Computer Systems Center
Lisle, Illinois

A WORD ABOUT TRADEMARKS . . .

The following AT&T registered trademarks are mentioned in this manual:

WE® 32001 Processor Module

WE® 32100 Microprocessor

WE® 32101 Memory Management Unit

WE® 32106 Math Acceleration Unit

WE® 321AP Microprocessor Analysis Pod

WE® 321DS Microprocessor Development System

WE® 321SD Development Software Programs

WE® 321SG Software Generation Programs

AT&T 3B2/3B5/3B15 Computers are trademarks of AT&T.

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

Select Code 305-000

AT&T

March 1986

AT&T 3B2/3B5/3B15 Computers
Assembly Language Programming Manual

The information contained herein is subject to change.

ACKNOWLEDGEMENTS

Prepared and published by
Document Development Organization – Microelectronics Projects Group
AT&T Network Systems, Morristown

for the

Microsystems Product Management
AT&T Technologies, Inc.

and the

AT&T Computer Systems Center
Lisle, Illinois

A WORD ABOUT TRADEMARKS . . .

The following AT&T registered trademarks are mentioned in this manual:

WE® 32001 Processor Module

WE® 32100 Microprocessor

WE® 32101 Memory Management Unit

WE® 32106 Math Acceleration Unit

WE® 321AP Microprocessor Analysis Pod

WE® 321DS Microprocessor Development System

WE® 321SD Development Software Programs

WE® 321SG Software Generation Programs

AT&T 3B2/3B5/3B15 Computers are trademarks of AT&T.

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

Printed in United States of America

Select Code 305-000

FOREWORD

This manual is intended for 3B Computer users with a need to program in assembly language. Emphasis is on the *WE* 32100 Microprocessor and its floating point support. The more general IS25 instructions are referenced in an appendix.

Data organization and storage, and *WE* 32100 Microprocessor (CPU) addressing modes are discussed prior to the instruction set. The instructions are grouped functionally into data transfer arithmetic, logical, program control, coprocessor, stack, and miscellaneous types. Mnemonics, op-codes, bytes, cycles counts, and effect on flag bits are tabulated for each instruction. Detailed descriptions of each instruction are given in the appendix. The CPU's operating system instructions are also referenced.

Assembler and disassembler options, along with a description of assembler directives and macro processing facilities, are presented. The *WE* 32106 Math Acceleration Unit assembly language instructions are detailed along with the alternative floating point emulation library of functions.

To obtain additional copies of this manual, Select Code 305-000, call:

□ **1-800-432-~~6000.~~** 6600

ASSEMBLY LANGUAGE PROGRAMMING MANUAL

FOR THE *AT&T* 3B2/3B5/3B15 COMPUTERS

CONTENTS

CHAPTER 1. INTRODUCTION

CHAPTER 2. ARCHITECTURE

CHAPTER 3. ASSEMBLY LANGUAGE STRUCTURE

CHAPTER 4. DATA ORGANIZATION

CHAPTER 5. ADDRESSING MODES

CHAPTER 6. INSTRUCTION SET

CHAPTER 7. USING THE as ASSEMBLER

CHAPTER 8. THE dis DISASSEMBLER

CHAPTER 9. OPERATING SYSTEM INTERFACE

CHAPTER 10. FLOATING POINT SUPPORT

APPENDIX A. *WE* 32100 MICROPROCESSOR INSTRUCTION SET

APPENDIX B. IS25 INSTRUCTION SET

LIST OF FIGURES

LIST OF TABLES

Chapter 1
Introduction

CHAPTER 1. INTRODUCTION

CONTENTS

1. INTRODUCTION

This chapter discusses the importance of assembly language programming, introduces the *WE* 32100 Microprocessor (CPU), and traces its development. An overview of the *WE* 32100 Microprocessor features is given along with a description of its development from the 8-bit 212 Series Microprocessor to the present 32-bit microprocessors.

1.1 OVERVIEW

Most programs for *AT&T* 3B Computers are written in C, a popular high-level programming language which was developed at AT&T Bell Laboratories. Programs written in C can be highly portable between different computers. Although, in some cases, the C language does not provide easy access to inherently machine dependent operations such as accessing external hardware registers. Often, a piece of software is so frequently used that it is crucial to obtain optimal performance from it. These situations are examples where a programmer should consider writing a program (or subroutine) in *WE* 32100 Microprocessor (CPU) assembly language for *AT&T* 3B2/3B5/3B15 Computers.

AT&T has developed a variety of computer products and is constantly enhancing both hardware and software. Each new generation brings with it additional capabilities that users may wish to take advantage of. For example, support of a math coprocessor (*WE* 32106 Math Acceleration Unit, MAU) was added to the *WE* 32100 CPU. In a few cases, it will be necessary for the assembly language programmer to know which generation of microprocessor (i.e., *WE* 32001 Processor Module or *WE* 32100 CPU) is being used on a particular machine. All software that works on the *WE* 32001 Processor Module will work on the *WE* 32100 Microprocessor. As indicated by the math coprocessor example, the converse is sometimes not true. Programmers can determine which microprocessor and support chips are used on their AT&T computer from their owners' manual or system administrator.

This manual describes the assembly language for the *WE* 32100 Microprocessor. A short description of earlier version 3B computer assembly language, IS25, which is source compatible with the *WE* 32100 Microprocessor, is included as a reference. Most programmers should, however, write in native *WE* 32100 Microprocessor assembly language.

Two methods of performing floating point operations (e.g., add, multiply) are described. One of them, the MAU Instruction Set (MIS), provides optimal floating point performance when it is known that a MAU will always be present in the system. The other method, which performs the operations via function calls, works whether or not a MAU is present.

The *WE* 32100 Microprocessor is a second-generation device with more speed and processing power than most minicomputers. Using complementary metal oxide semiconductor (CMOS) twin-tub technology, over 180,000 transistors have been placed onto the one-quarter inch silicon square comprising the device.

The CPU's design was based on its immediate predecessor, the 32-bit *WE* 32001 Processor Module. Both microprocessors have 32-bit data and address buses. The 32-bit data bus allows fetching 32-bits of data in one memory fetch cycle. This significantly reduces memory retrieval time, which is the limiting factor in most microprocessors. The 32-bit address bus allows for a directly addressable 4.3 billion byte address space. Since all on-chip registers are also 32-bits wide, 32-bits of data can also be processed in one execution cycle.

Although the *WE* 32100 Microprocessor is a faster and smaller version of the *WE* 32001 Processor Module, it is a true second generation 32-bit machine. The *WE* 32100 Microprocessor has a 64-word, 32-bit high-speed cache memory not available on the earlier processor, improved pipelining capability, and a new I/O controller that supports both distributed processing and coprocessor interfaces. The on-chip instruction cache represents a technological first for any 32-bit processor.

Additionally, both 32-bit processors were designed to be an efficient execution vehicle for the *UNIX* Operating System and the C programming language. The C compiler and associated software tools make it relatively easy to write optimum code for any application.

1.2 DEVELOPMENT

The AT&T family of 32-bit processors are direct descendants of the 8-bit 212 Series Microprocessor developed at AT&T Bell Laboratories in the mid-1970s. The 212 Series Microprocessor consisted of 10,000 transistors and performed approximately 200,000 instructions per second using a 3 MHz clock. This 8-bit microprocessor was unique in its use of sixteen general-purpose memory-addressed accumulators and its fabrication in CMOS technology. CMOS, which has since has become the technology of choice for current state-of-the-art microprocessors, was developed early in the 1970s at Bell Laboratories for use in all AT&T microprocessors. CMOS devices use significantly less power than equivalent devices designed in n-channel mos (NMOS), are highly immune to signal interference, and can operate over wide ranges of voltage and temperature. These characteristics are especially important for designing the higher density devices typified by 32-bit microprocessors.

The 212 Series Microprocessor was followed by the single-chip, 4-bit 301 Series Microcomputer. This latter device used 30,000 transistors. The significance of the 301 Series, however, was the introduction and use of AT&T Bell Laboratories internally developed, computer-aided, design technologies in its design and development. The design, development, testing, and introduction of a new microprocessor typically requires two to three years; the computer-aided design and testing cycle developed for the 301 Series would enable the *WE* 32001 Processor Module to be developed and operational in less than half of that time.

The *WE* 32001 Processor Module required significantly higher levels of software design, development, and hardware technology than the 301 Series. The design of this first 32-bit microprocessor involved the sophisticated innovations and refinements, in both the CMOS technology and the computer-aided design (CAD) techniques, that were previously used on the 301 Series. As a result, the *WE* 32001 Processor Module was developed and became successfully operational in thirteen months.

The computer-aided design and development tools used on the *WE* 32001 Processor Module had an equally significant impact on the introduction of the *WE* 32100 Microprocessor. The *WE* 32100 Microprocessor was developed and fabricated in eleven months, with the first device containing only one minor layout error in over 180,000 transistors.

The chapters that follow describe the architecture of the *WE* 32100 Microprocessor and the assembly language instruction set of the processor as it is used within the 3B family of AT&T computers.

Chapter 2
Architecture

CHAPTER 2. ARCHITECTURE

CONTENTS

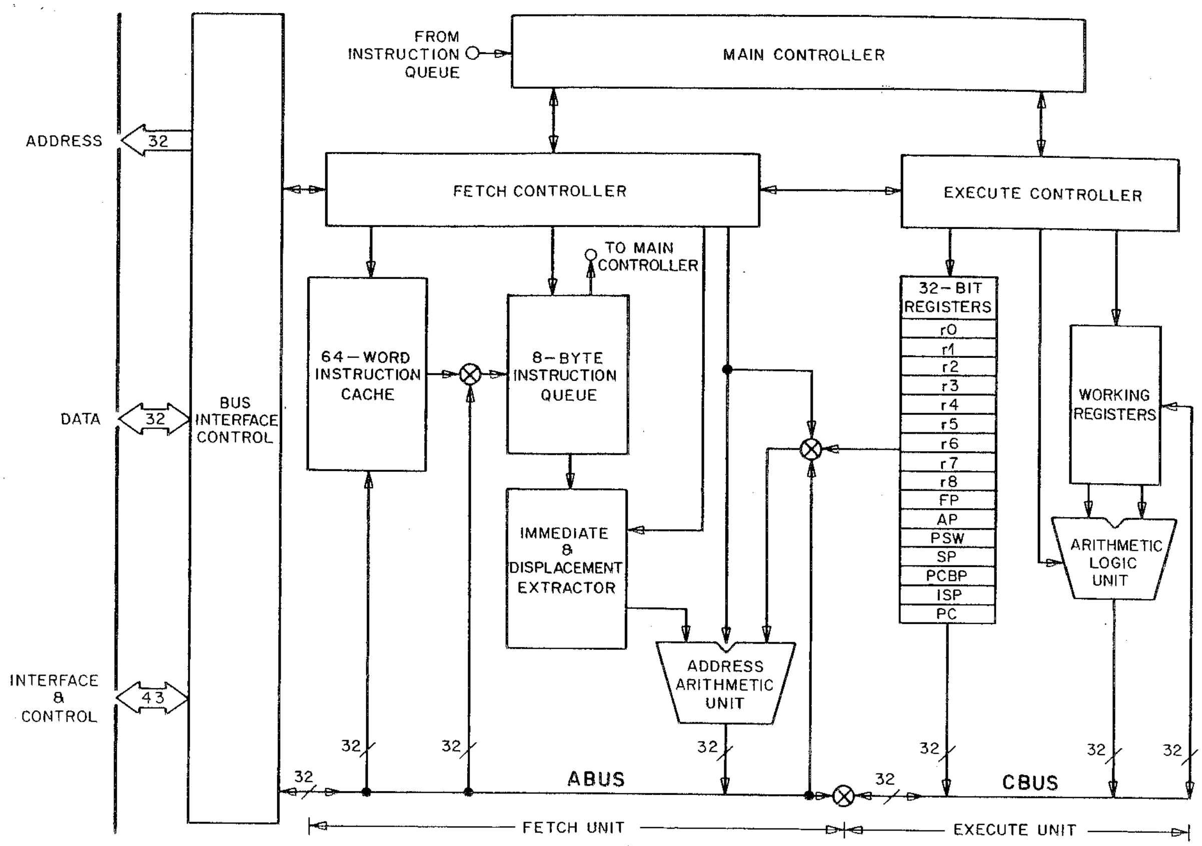

Figure 2-1. ***WE* 32100 Microprocessor Block Diagram**

2. ARCHITECTURE OF THE *WE* 32100 MICROPROCESSOR

In this chapter we will look at the architecture of the *WE* 32100 Microprocessor and its internal register set. A block diagram of the *WE* 32100 Microprocessor illustrating its four major sections: main controller, fetch unit, execute unit, and bus interface control, is shown on Figure 2-1.

The Main Controller is responsible for directing the actions of the Fetch and Execute Controllers as instructions are executed.

The Fetch Unit is responsible for fetching all instructions and data. Although the operation of this unit is transparent to the microprocessor user, it contains unique features which significantly enhance the performance of the *WE* 32100 Microprocessor. One of these features is a 64-word instruction cache, which stores prefetched instructions from memory. The prefetched instructions are retrieved from memory simultaneous with instruction execution (a technique known as pipelining). Thus, the normal suspension of execution, while the processor waits for an instruction to be read from memory, is avoided when the next instruction is available in the cache.

The Execution Unit provides all of the features of the microprocessor which are directly user accessible. This unit performs all arithmetic, logical, data-movement, and program control instructions. Contained in the Execution Unit are the sixteen 32-bit user accessible registers, consisting of nine general-purpose registers (r0—r8), and seven special-purpose registers (r9—r15).

In addition to supporting an extremely powerful assembly language, the registers in the *WE* 32100 Microprocessor were also designed for the efficient support of procedure-oriented high level languages, such as C, and process-oriented operating systems, such as the *UNIX* Operating System.

Although all of the sixteen registers are available to assembly language programmers, it is useful to separate the register set into three groups: the Assembly Level support group, the High-Level Language support group, and the Operating System support group. These groups are illustrated on Figure 2-2.

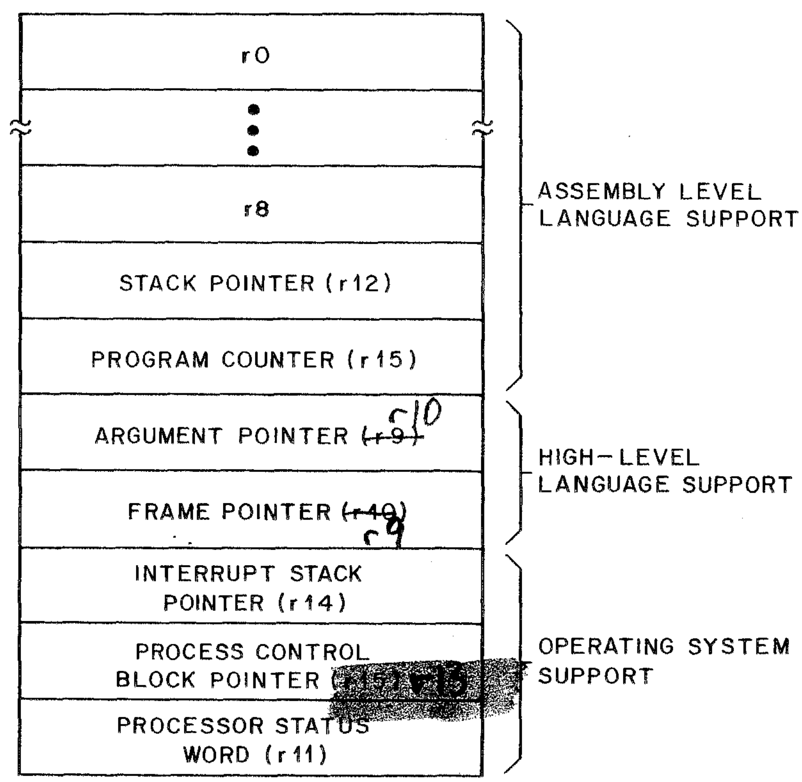

Figure 2-2. The *WE* 32100 Microprocessor Register Set

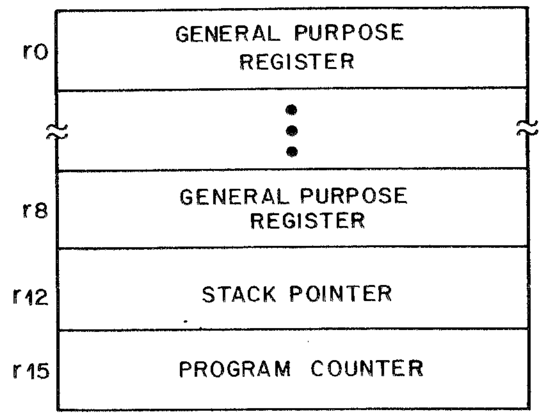

Figure 2-3. Assembly Level Support Group

2.1 ASSEMBLY LEVEL SUPPORT GROUP

The assembly level support registers consist of nine general-purpose registers, a stack pointer, and a program counter. This set of registers, including condition flags, is typically associated with an assembly language programming model of a microprocessor. In the *WE* 32100 Microprocessor the condition flags, which are indicators of the processor's current status, are contained within the processor status word register.

The nine general-purpose registers are referred to as r0 through r8, respectively. These registers can be used with all arithmetic, data transfer, logical, and program control assembly instructions. Additionally, registers r0, r1, and r2 are used in both string manipulation and transfer instructions and, by convention, for returning values from a called C language program. The string manipulation and transfer instructions that use registers r0, r1, and r2 include the block move (MOVBLW), string copy (STRCPY), and string end (STREND) instructions (see **6. Instruction Set**).

The *Stack Pointer* (SP), r12, contains the current 32-bit address of the top of the current execution stack. As illustrated on Figure 2-4, the stack pointer points to the next available memory location that can be used. A PUSH instruction immediately stores its operand at the current memory address contained in the stack pointer. The stack pointer is then incremented by the site of the PUSHed operand. Thus, the stack "grows" into increasing memory address space. A POP instruction first decrements the stack pointer by the site of the POPed operand (to point to the last pushed operand) and then fetches the data from the top of the stack.

The *Program Counter* (PC), r15, contains the 32-bit memory address of the currently executing instruction or, upon completion, the starting address of the next instruction to be executed. The PC is referenced by all program control instructions and all function calls and returns.

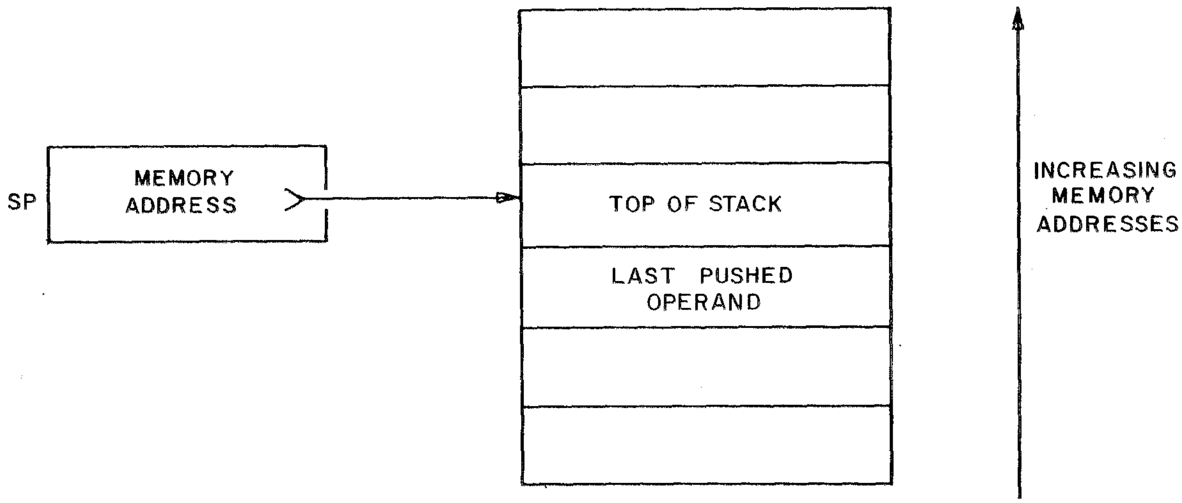

Figure 2-4. The *WE* 32100 Microprocessor Stack

2.2 HIGH-LEVEL LANGUAGE SUPPORT GROUP

The *Frame Pointer* and *Argument Pointer* constitute the high-level language support group register set. Although these two registers may be accessed and used as general-purpose assembler registers, they are typically used in association with registers r0, r1, and r2 for passing, holding, and returning high-level language variables and arguments.

These registers perform the following functions:

The *Frame Pointer* (FP), r9, points to the beginning location in the stack where the local variables of the currently running program, procedure, or function are stored. The frame pointer is implicitly changed by the save registers (SAVE) and restore registers (RESTORE) assembly instructions.

The *Argument Pointer* (AP), r10, points to the beginning location in the stack where arguments passed into the currently running program, procedure, or function have been pushed. The ap is implicitly affected by procedure call (CALL) and return (RET) assembly instructions.

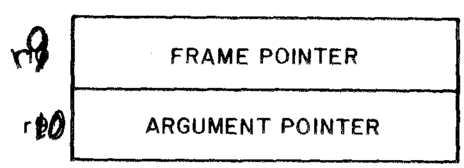

Figure 2-5. High-Level Language Register Support Group

2.3 OPERATING SYSTEM SUPPORT GROUP

The Processor Status Word, Process Control Block Pointer, and Interrupt Stack Pointer were designed to facilitate an efficient operating system interface. These three registers, therefore, are referred to as the operating system support group.

The three registers forming this group perform the following functions:

The *Processor Status Word* (PSW), r11, contains status information about the microprocessor and the current process. Additionally, the PSW contains four condition code flags used by assembly language transfer-of-control instructions. In general, the PSW changes as a whole only when a process switch occurs and can only be written in an operating system mode.

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 an operating system mode.

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 the microprocessor is in an operating system mode.

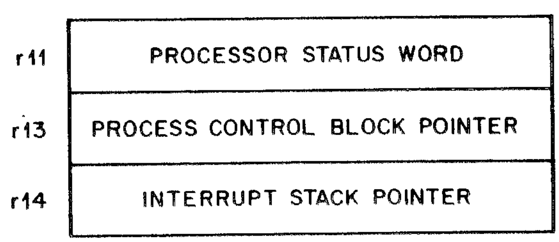

Figure 2-6. Operating System Register Support Group

Chapter 3

Assembly Language Structure

CHAPTER 3. ASSEMBLY LANGUAGE STRUCTURE

CONTENTS

3. ASSEMBLY LANGUAGE STRUCTURE

This chapter describes the assembly language, syntax, and semantics supported by the assembler (as) provided for the 3B2/3B5/3B15 Computers. All 3B2/3B5/3B15 Computers have the same basic instruction set. But, some models have additional instructions due to the advanced features of the *WE* 32100 Microprocessor architecture. Therefore, the discussion on the microprocessor instruction set will be based on the *WE* 32100 Microprocessor instruction set. Contained in Appendix C are some assembly language programming examples which conform to the syntax and semantics described in this chapter.

The basic actions of evaluation, assignment, and control of evaluation order are specified by statements. Statements are either microprocessor instructions, assembler directives, or IS25 instructions.

The data types supported by the assembly language are byte, halfword, word, single, double, double extended, and bit field. A byte is an 8-bit quantity; a halfword is a 16-bit quantity; a word is a 32-bit quantity; a single is a 32-bit floating point quantity; a double is a 64-bit floating point quantity; a double extended is a 96-bit floating point quantity; and a bit field is a sequence of 1 to 32 bits. Detailed information on the data types can be found in **4. Data Organization**.

The instruction set provides that bytes, halfwords, words, singles, doubles, and double extendeds can be interpreted as either signed or unsigned quantities for arithmetic or logical operations.

3.1 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), which is equivalent to control -J (line-feed). 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. Assembler Directives are discussed in detail in **7.3 Assembler Directives**.
- *WE* 32100 Microprocessor Instruction — a mnemonic representation of an executable machine instruction. It consists of an operation code followed by zero or more operands. *WE* 32100 Microprocessor instructions are discussed in detail in Appendix A.
- IS25 Instruction — a statement that maps into one or more executable microprocessor instructions. IS25 instructions are discussed in detail in Appendix B.
- Empty — a statement that contains only spaces, tabs, or a comment. It signifies nothing to the assembler, but is often used to enhance program readability.

Operation codes (or mnemonics) 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.

In the order shown, each nonempty line of code is made up 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 **3.3 Symbols**. An unlabeled statement MUST have a space, tab, or pound sign (#) in the first character position.
- A **mnemonic** may be placed on any statement. The mnemonic consists of a *symbol* that begins after any white space at the beginning of a statement or after a label. The mnemonic defines an assembler directive or a machine operation (either processor or IS25 instruction).
- One or more **operands** may be placed on a statement containing a mnemonic. An operand, in the case of a machine instruction, defines the addressing modes of source and destination operands. These type of operands are further discussed in **4.1 Data Types** and **5. Addressing Modes**. Operands supplied with an assembler directive are used by the assembler to execute the command issued by the assembler directive. These operands are discussed in **7.3 Assembler Directives**.
- A **comment** may be inserted at the end of any statement by preceding the comment with a pound sign (#) or may be on a line by itself by inserting a pound sign in the first character position. 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. Multiline 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 line 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 an IS25 and processor instruction, respectively.

NEED DOT

Label	Mnemonic	Operand(s)	Comment
	globl	prefix	#Assembler Directive
main:	save	&1	#IS25 Instruction
	ADDW2	%r1,%sp	#Processor instruction

3.2 EXECUTABLE INSTRUCTIONS

Mnemonics for processor instructions use uppercase letters and IS25 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 IS25 instructions must be coded in lowercase.

Be careful when switching between processor and IS25 instructions. Although the mnemonics are identical in many cases, the operations are not. For example, the IS25 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 and often faster than the IS25 instruction set. IS25 instructions can be portable to earlier version 3B computers, while processor instructions can not.

3.3 SYMBOLS

Symbols are names recognized by the assembler. They always have a value and type, either specified explicitly by an assignment statement (see **7.3 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 used as labels, mnemonics, or operands (in some cases). 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. The .text section contains executable instructions.

3. **.data** This symbol has type DATA and is used to label the beginning of the **.data** section for the program being assembled. The .data section contains initialized variables.

4. **.bss** This symbol has type BSS and is used to label the beginning of the **.bss** section for the program being assembled. The .bss section contains uninitialized variables.

3.3.1 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 file or module 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 file or module currently being assembled, but is defined in some other program.

3.3.2 Assigning Values and Types to Symbols

There are two ways to assign a value and a 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 (see **7.3.3 Assignment Pseudo Operations**). An arbitrary value and type can be assigned with this directive.

3.4 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 either to assembler directives or to machine instructions, as maybe 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 difference in address 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.

3.4.1 Constants

A constant is an object of ABSOLUTE type and fixed value. The size and appropriate number of digits are controlled by the generation of 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 0x or 0X. Examples of hexadecimal constants are:

  0x3f   0X9aC   0xabcd   0XFE

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 to conform to the IEEE standard for binary floating-point arithmetic.

3.4.2 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.

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, PC, (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 problematic for any conditional branch instructions; the instructions wait until the condition codes are valid before they are then tested.

Chapter 4
Data Organization

CHAPTER 4. DATA ORGANIZATION

CONTENTS

4. DATA ORGANIZATION

This chapter describes the data types, data organization, and data storage supported by the *WE* 32100 Microprocessor.

4.1 DATA TYPES

The data types supported by the *WE* 32100 Microprocessor are byte, halfword, word, floating point (word, double word, and double extended word), and bit field data. The instruction set provides that bytes, halfwords, and words can be interpreted as either signed or unsigned quantities.

A *byte* is an 8-bit quantity that may appear at any address. Bits are numbered from right to left within a byte, starting with zero, the least significant bit (LSB), and ending with 7, the most significant bit (MSB), as illustrated on figure 4-1.

A *halfword* is a 16-bit quantity that may appear at any address divisible by two. Bits are numbered from right to left starting with zero, the LSB, as illustrated on Figure 4-2.

A *word* is a 32-bit quantity. Data words may appear at any address divisible by four. Bits are numbered right to left starting with zero, the LSB, as illustrated on Figure 4-3.

Floating Point data types may appear at any address in memory divisible by four. Figure 4-4 illustrates the floating point data types supported by the assembler.

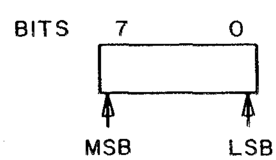

Figure 4-1. Byte Data

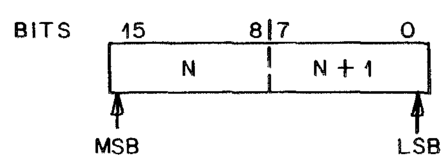

Figure 4-2. Halfword Data

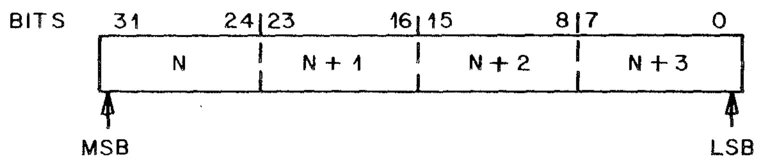

Figure 4-3. Word Data

Each of these four data types may be interpreted as either a signed or unsigned quantity, with signed data represented in 2's complement form.

A *bit field* is a sequence of 1 to 32 bits extracted from a byte, halfword, or a word. The bit field is determined from the address of the word containing the field, an offset, and a width. The offset, from 0 to 31, identifies the starting bit in the word containing the bit field. This bit becomes the least significant bit of the selected field. The width, a number from 0 to 31 specifies the size of the field. The number of bits in the extracted field is one more than the width value. Figure 4-5 illustrates a bit field extracted from a word using an offset of six and a width of nine. Notice that the extracted field contains ten bits, one more than the width.

Bit fields do not extend across word boundaries. If the selected width requires bits beyond the most significant bit of the word being used, the extraction of bits continues by wrapping around to the least significant word bits.

Bit	31	30 23	22 0
Field	Sign	Exponent	Fraction

A. Single Precision Floating Point Data Type

Bit	63	62 52	51 0
Field	Sign	Exponent	Fraction

B. Double Precision Floating Point Data Type

Bit	85 80	79	78 64	63	62 0
Field	Unused	Sign	Exponent	J	Fraction

C. Double Extended Floating Point Data Type

Figure 4-4. Floating Point Data Types

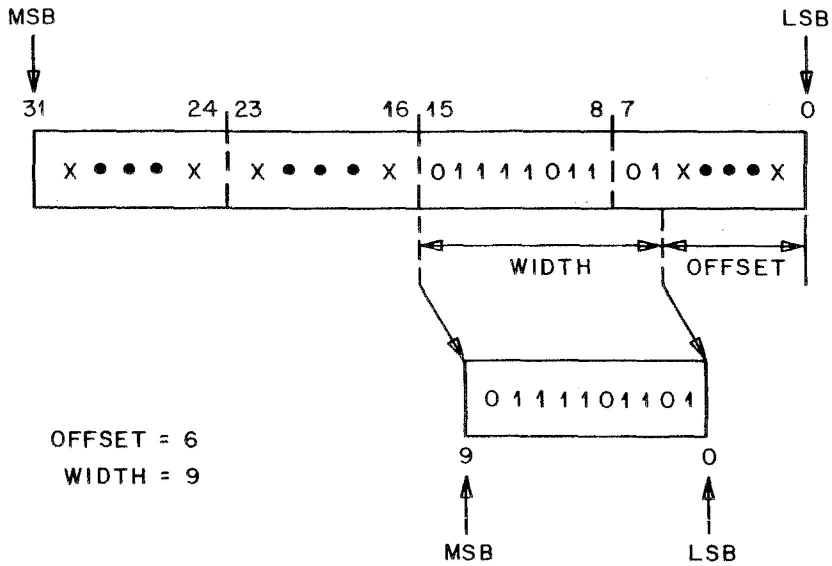

Figure 4-5. Extraction of a Bit Field

4.2 DATA STORAGE IN MEMORY

Figure 4-6 illustrates the storage of word data in memory. As illustrated, the word 0x12345678 is stored with the lower-order bytes at higher-order addresses. All data stored in memory follows this format. For example, the halfword data 0xEEFF would be stored in memory with the lower-order byte, 0xFF, at the next higher-byte address than the location containing the byte 0xEE.

4.3 REGISTER DATA STORAGE

All data stored in a register is a full 32 bits, regardless of the instruction or data type. For all CPU operations, including register storage, the *WE* 32100 Microprocessor reads in the correct number of bits for the operand and extends the data automatically to 32 bits. Halfword operands and signed data are sign extended to 32 bits. In sign extension, the value of the most significant bit is replicated to fill the high-order bits. When storing byte operands or unsigned data into a register, zero extension is used. In zero extension, the high-order bits are filled with zeros.

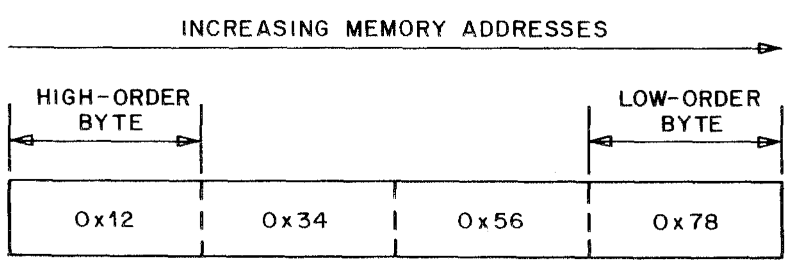

Figure 4-6. Word Storage in Memory

Intermediate results of all operations in the CPU are always 32 bits. If the results of an operation are stored in a register, the processor writes all 32 bits to the register.

When a register is specified as the source of a byte operand, the low-order 8 bits (bits 0—7) of the register are fetched and zero extended to 32 bits. The zero extension may be

changed to a sign extension using an expanded operand type addressing mode (this addressing mode is described in **5. Addressing Mode**). When a register is used as the source of a halfword operand, the low-order 16 bits (bits 0—15) of the register are fetched and sign extended to 32 bits. Again, the type of extension may be changed to zero extension using an expanded operand type addressing mode.

4.4 INSTRUCTION STORAGE IN MEMORY

Instructions may appear at any byte address in memory, and are stored as a one- or two-byte opcode followed by up to four operands. Figure 4-7 illustrates the general format of an assembly instruction as it is stored in memory. Each individual operand shown on Figure 4-7 consists of a descriptor byte, followed by up to four bytes of data (see Figure 4-8).

The descriptor byte defines an operand's addressing mode and register fields, which are covered in the next chapter. Immediate data stored within an instruction is stored with lower-order bytes located at lower-order addresses. For example, the value 0x12345678 would be stored within an instruction as illustrated on Figure 4-9.

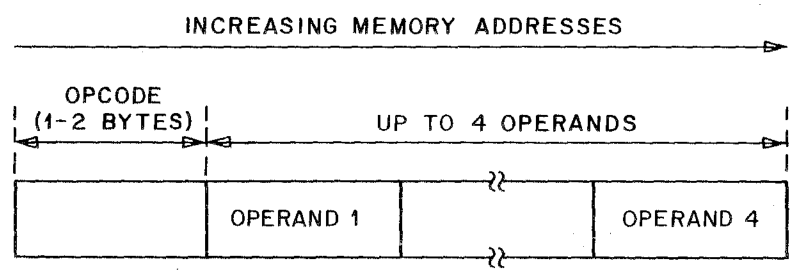

Figure 4-7. Instruction Storage in Memory

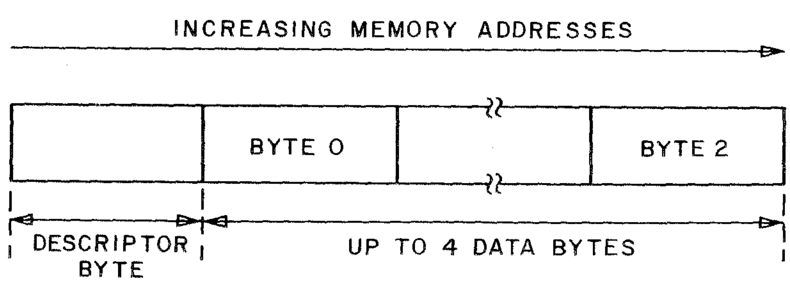

Figure 4-8. Operand Format

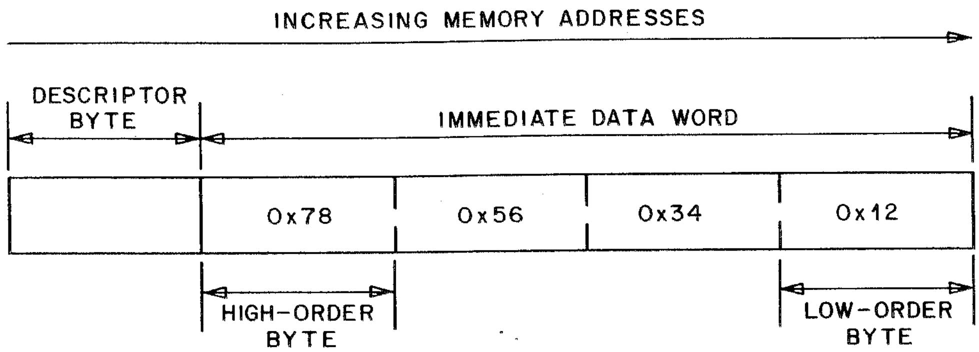

Figure 4-9. Word Storage Within an Instruction

Notice that the storage of data within an instruction, as shown on Figure 4-9, is the reverse of the storage of data within a memory location as illustrated on Figure 4-6.

Chapter 5
Addressing Modes

CHAPTER 5. ADDRESSING MODES

CONTENTS

5. ADDRESSING MODES

In this chapter, we will look at addressing modes for the *WE* 32100 Microprocessor, their assembly language coding, and their storage in memory.

An assembly language instruction for the *WE* 32100 Microprocessor consists of a mnemonic, such as ADDW, MOVH, INCB, followed by up to four operands. Each operand is physically located as immediate data in either one of the microprocessor's registers, a memory location, an input-output port, or directly within the instruction. The operand written in the assembly language instruction must provide sufficient information for the actual operand to be located by the microprocessor. The information provided by the assembly language instruction to specify an operand's address is called addressing mode data.

Table 5-1 provides a partial listing of the microprocessor's basic addressing modes. A complete description of addressing modes is presented in Table 5-2.

Table 5-1. Basic Addressing Modes

Mode	Syntax	Example
Register	%reg	%r2
Register Deferred	(%reg)	(%r2)
Register Displacement	expr(%reg)	6(%r2)
Register Displacement Deferred	*expr(%reg)	*6(%r2)
Immediate	&expr	&0x1234
Absolute	$expr	$0x2E54
Absolute Deferred	*$expr	*$0x2E54

Notes:
1. reg represents one of the microprocessor's registers (r0—r15).
2. expr is an expression that evaluates to either a byte, halfword, or word value.

An assembly language instruction is stored in memory as a one- or two-byte opcode followed by up to four operands. Figure 5-1 illustrates the memory storage format of an assembly instruction previously described in Chapter 4. Recall that each operand shown in Figure 5-1 consists of a descriptor byte, followed by up to four bytes of data (see Figure 5-2).

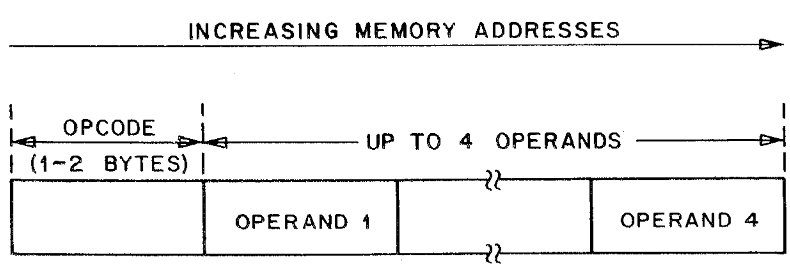

Figure 5-1. Instruction Format

The descriptor byte defines an operand's addressing mode and register field. Figure 5-3 illustrates the format of the descriptor byte, which consists of two 4-bit fields.

The register field, denoted as rrrr, consists of bits 0 through 3 of the descriptor byte, and contains the number of a register, 0 through 15. The mode field, denoted as mmmm, consists of the four higher-order bits of the descriptor byte, bits 4 through 7. This field contains an address mode number, 0 through 15. Table 5-2 lists all mode field values (0—15) and their corresponding addressing modes.

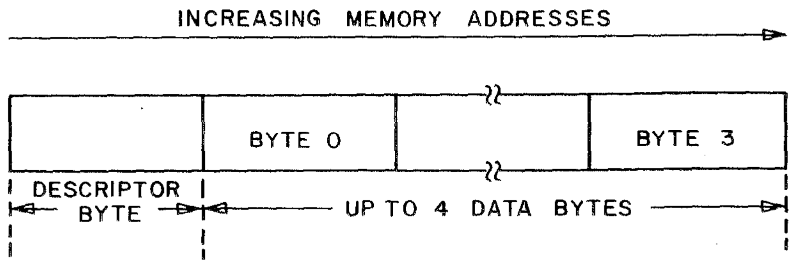

Figure 5-2. Operand Format

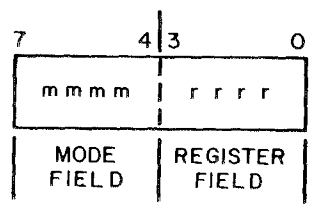

Figure 5-3. Descriptor Byte Format

Table 5-2. The *WE* 32100 Microprocessor Addressing Modes

Mode Field Value	Addressing Mode	Description
0—3	Literal	The register field bits are concatenated with the two low-order mode field bits to form an unsigned 6-bit immediate data.
4	Register	The operand is contained in one of the 16 registers. If register 15 is specified in the register field, this becomes the word immediate mode.
5	Register Deferred	The register specified in the register field contains the operand's address. If register 15 is specified in the register field, this becomes the halfword immediate mode.
6	FP Short Offset	The FP (register 9) is implicitly referred to by this mode. Register field bits are used as an offset and are added to the FP to form the operand's address. This addressing mode is an optimized case of the register deferred mode, produced by the assembler.

Table 5-2. The *WE* 32100 Microprocessor Addressing Modes (Continued)

Mode Field Value	Addressing Mode	Description
7	AP Short Offset	The AP (register 10) is implicitly referred to by this mode. Register field bits are used as an offset and are added to the AP to form the operand's address. If register 15 is specified by the register field, this mode becomes the absolute mode. The four bytes following the descriptor byte contain the operand's address. This addressing mode is an optimized case of the register deferred mode, produced by the assembler.
8	Word Displacement	The four bytes following the descriptor byte are added to the contents of the register specified in the register field. The sum forms the address of the operand.
9	Word Displacement Deferred	The four bytes following the descriptor byte are added to the contents of the register specified in the register field. The sum forms the address of a pointer. The operand's address is contained within the pointer.
A	Halfword Displacement	The two bytes following the descriptor byte are added to the contents of the register specified in the register field to form the operand's address.
B	Halfword Displacement Deferred	The two bytes following the descriptor byte are added to the contents of the register specified in the register field. The sum forms the address of a pointer. The operand's address is contained within the pointer.
C	Byte Displacement	The byte following the descriptor byte is added to the contents of the register specified in the register field to form the operand's address.
D	Byte Displacement Deferred	The byte following the descriptor byte is added to the contents of the register specified in the register field. The sum forms the address of a pointer. The operand's address is contained within the pointer.
E	Expanded Operand	This mode is used to modify the data type of an operand. If register 15 is specified in the register field, this becomes the absolute deferred mode.
F	Negative Literal	The register field bits are concatenated with the mode field bits to form a negative literal, in the range −1 to −16.

We will now consider each of the microprocessor's addressing modes and see how they are coded within an operand's descriptor byte. It should be noted that certain branch instructions, and all coprocessor words do not require addressing mode information and therefore do not use a descriptor byte.

5.1 REGISTER MODE

Any operand directly located in one of the microprocessor's registers is accessed using the register address mode. This mode is indicated in assembly language with the percent symbol (%).

For example, the instruction INCW % r2 causes the 32-bit contents of register r2 to be incremented by one.

The general syntax, mode, and register fields used to signify the register addressing mode are:

syntax: % rn where n is a register number
mmmm: 4
rrrr: 0 to 14

Thus, the instruction INCW % r2 is stored in memory as illustrated on Figure 5-4.

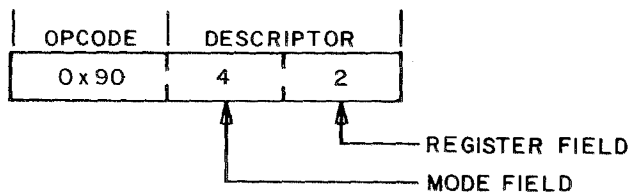

Figure 5-4. Register Mode Example

5.2 REGISTER DEFERRED MODE

Deferred addressing mode involves indirect addressing using pointers. A pointer is either a register or memory location containing an address. Figure 5-5 illustrates the relationship between the address contained in a pointer and the operand ultimately obtained. The term *deferred* is used to describe this procedure because the operand finally obtained is deferred, or delayed, by first going to the pointer for an address. The address contained in the pointer is then used to access the desired operand.

When deferred addressing is used and the pointer is one of the microprocessor's registers, the addressing mode is referred to as a register deferred mode. This addressing mode is designated in assembly language by using parentheses around the pointer register.

For example, the instruction MOVW (% r2),% r3 causes the CPU to regard the data in register r2 as an address. The contents of the memory location having this address will be copied into register r3. Notice that this instruction uses two operands and each operand has its own addressing mode. Although a register deferred mode was used for the source operand and a register mode was used for the destination operand, any other valid addressing modes could have been used.

The general syntax, mode, and register fields for a register deferred mode operand are:

syntax: (% rn) where n is a register number
mmmm:5
rrrr: 0 to 10, 12 to 14

Using this information, the instruction MOVW (% r2),% r3 is stored in memory as shown on Figure 5-6.

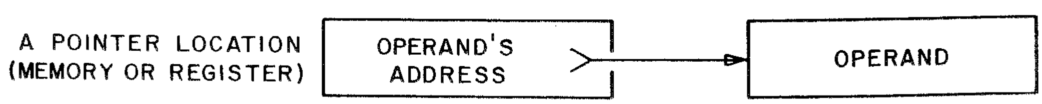

Figure 5-5. Deferred Addressing Using a Pointer

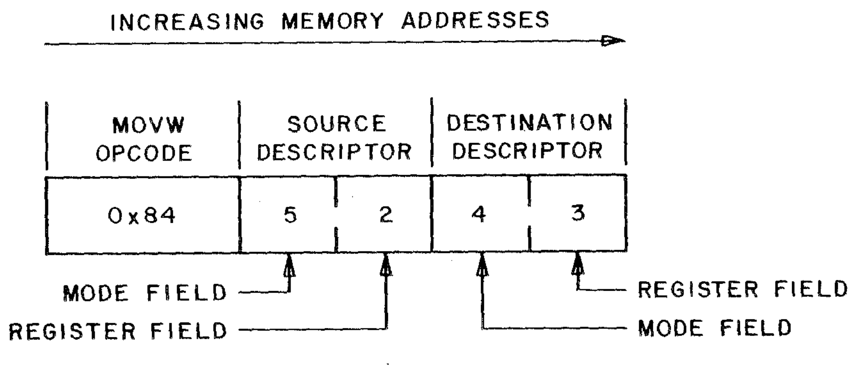

Figure 5-6. Register Deferred Mode Example

5.3 DISPLACEMENT MODE

The displacement mode forms an operand's address by adding an offset to the contents of a *WE* 32100 Microprocessor register. For example, the instruction MOVW 0x30(% r2),% r3 copies the contents of a memory location into register r3. The source operand's memory address is calculated as the contents of register r2 plus an offset of 0x30. Figure 5-7 illustrates the result of this MOVW instruction.

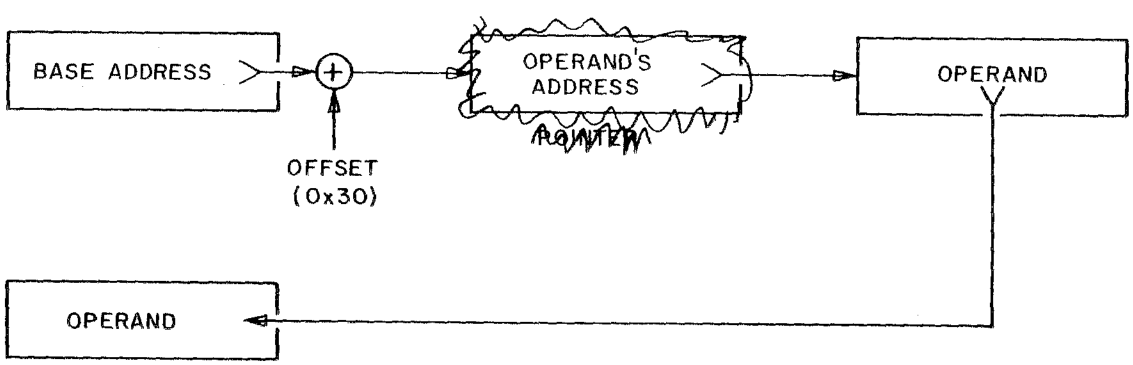

Figure 5-7. Example of MOVW 0x30(% r2),% r3

The general syntax, and valid register fields for a displacement mode operand are:

syntax: offset(% rn) where n is a register number
mmmm: 8, 10, or 12 (word, halfword, or byte offset)
rrrr: 0 to 10, 12 to 15

Using the appropriate mode and register fields, the instruction MOVB 0x30(% r2),% r3 is stored in memory as shown on Figure 5-8.

The offset used in the displacement mode may be either a byte (8-bits), halfword (16-bits), or word (32-bits), or an expression yielding such a value. 2's complement, negative offsets are also valid. Negative byte and halfword offsets are first sign-extended to 32 bits before being used to obtain the operand's final address. This sign extension converts a negative byte or halfword into its equivalent 32-bit counterpart.

When the displacement mode is used with registers FP (frame pointer) and AP (argument pointer), only a short offset between 0 and 14 may be used. This facilitates storage of a shortened instruction format in memory. The mode fields, when the frame and argument registers are used in the displacement mode, are 6 and 7, respectively. The short offset (0—14) is stored in the register field and extra bytes for an offset are not included in the stored instruction.

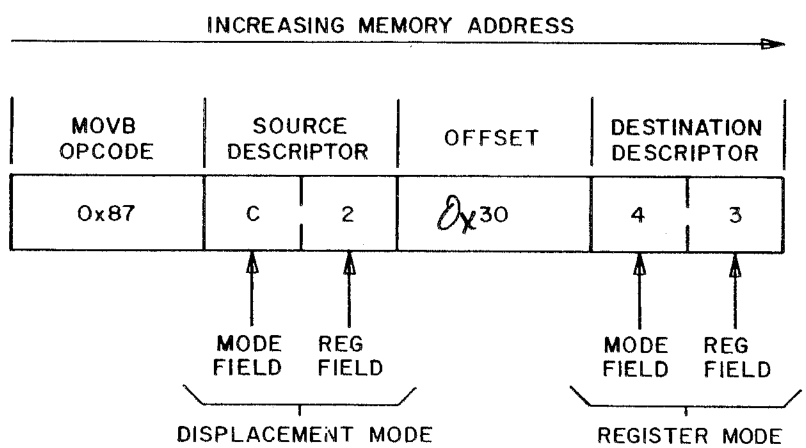

Figure 5-8. A Displacement Mode Source Operand

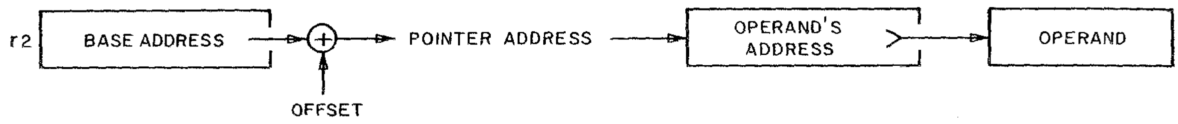

Figure 5-9. Deferred Displacement Addressing

5.4 DEFERRED DISPLACEMENT MODE

The deferred displacement mode uses the contents of the address calculated in the displacement mode as a pointer to the desired operand. Consider the example shown on Figure 5-9. For a typical displacement mode, the operand would be located in the first memory address calculated. In deferred displacement mode, the contents of this location are used as the address of the desired operand.

The deferred displacement mode is indicated to the assembler by the use of an asterisk before the offset.

For example, the instruction INCW *0x30(% r2) adds one to the contents of a memory location whose address is contained within a pointer. The address of the pointer is the contents of register r2 plus 0x30.

The general syntax, mode field, and register field for a deferred displacement mode operand is:

syntax: *expr(% rn)
mmmm: 9, 11, or 13 (word, halfword, or byte offset)
rrrr: 0—10, or 12—15

Using this information, the instruction MOVB *0x30(% r2),% r3 is stored in memory as illustrated on Figure 5-10.

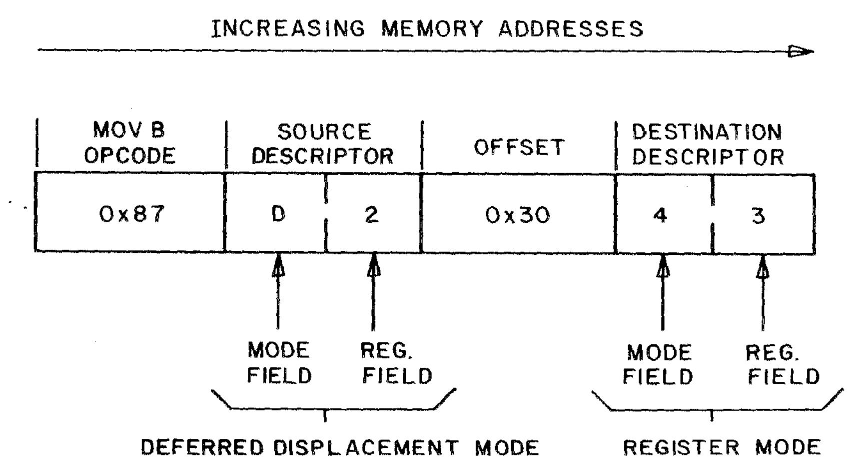

Figure 5-10. A Deferred Displacement Mode Source Operand

5.5 IMMEDIATE MODE

In the immediate addressing mode, the operand is contained within the instruction. The ampersand symbol is used to indicate this addressing mode to the assembler.

For example, the instruction MOVB &0x50,% r6 copies the immediate data, 0x50, into register r6. The & symbol signifies that the data immediately following is to be treated as immediate data. The % symbol, as should now be familiar, indicates that the register mode is being used for the destination operand.

The general syntax, valid mode and register fields for the immediate addressing mode are:

syntax: &data (data = 8-, 16-, or 32-bits)
mmmm: 4, 5, or 6
rrrr: 15

A mode field of 4 indicates that the immediate data is 32-bits long, while mode fields of 5 and 6 are used for 16-bit and 8-bit immediate data, respectively. Figure 5-11 illustrates the storage of the instruction MOVW &0x12345678,% r2 in memory. This instruction causes the immediate data, 0x12345678, to be placed into register r2.

Notice on Figure 5-11 that the immediate data is stored in memory with lower order bytes stored at lower order addresses. This is true for all immediate data; for example, the 16-bit immediate data 0xABCD would be stored as CDAB, with the byte containing CD stored at the immediately lower address than the byte containing AB.

The immediate mode also has a short storage form for positive immediate data between 0 and 63, and negative data between −1 and −16. In these two cases, the immediate data is stored directly within the descriptor byte.

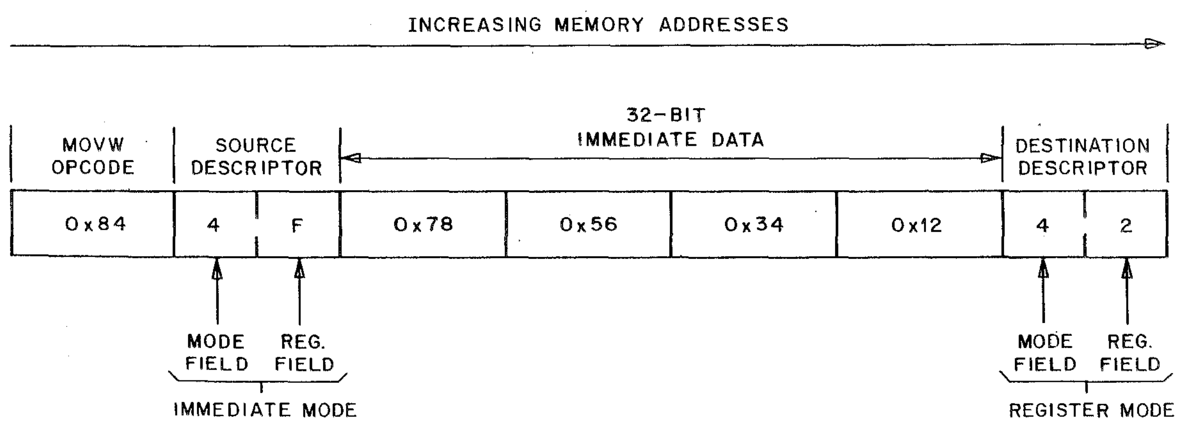

Figure 5-11. A 32-bit Immediate Source Operand

5.6 ABSOLUTE MODE

In this mode, the address of the desired operand is contained directly within the instruction. The dollar symbol is used to indicate this addressing mode to the assembler.

For example, the instruction MOVB $0x2E04,% r0 moves the byte starting at location 0x2E04 into register r0. The general syntax, mode, and register fields for the absolute address mode are:

> *syntax*: $expr (expr must yield to a byte, halfword, or word)
> mmmm: 7
> rrrr: 15

Thus, the instruction MOVB $0x2E04,% r0 is stored in memory as shown on Figure 5-12.

As illustrated on Figure 5-12, the memory address is stored as a 32-bit address with lower-order bytes stored in lower order memory addresses.

5.7 ABSOLUTE DEFERRED MODE

In the absolute deferred mode, the address contained within the instruction is used as a pointer to a word containing the address of the operand. As in all deferred modes, an asterisk is used to indicate deferred addressing to the assembler.

For example, the instruction MOVB *$0x2E04,% r0 uses the data contained within memory location 0x2E04 as the address of the source operand. The general syntax, mode, and register fields for this deferred mode is:

> *syntax*: *$expr (expr must yield to a byte, halfword, or word)
> mmmm: 14
> rrrr: 15

Thus, the instruction MOVB *$0x2E04,% r0 is stored in memory as illustrated on Figure 5-13.

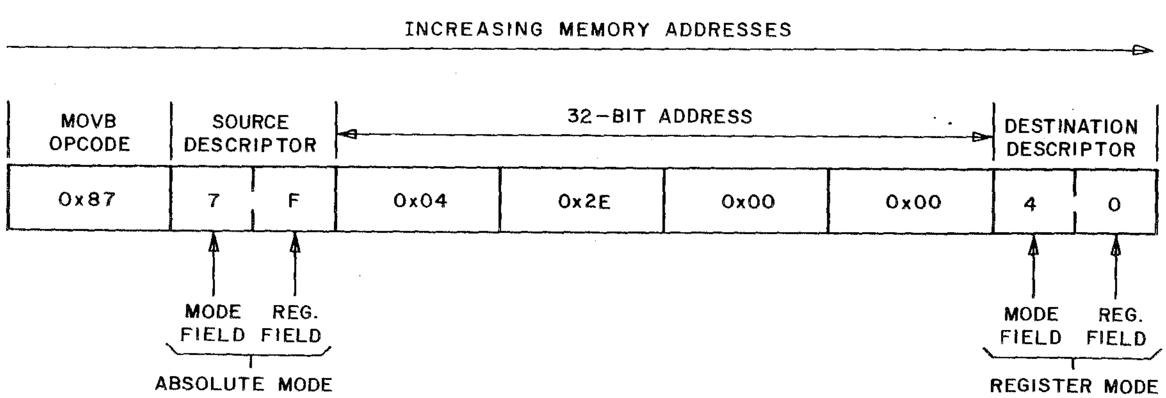

Figure 5-12. An Absolute Mode Source Operand

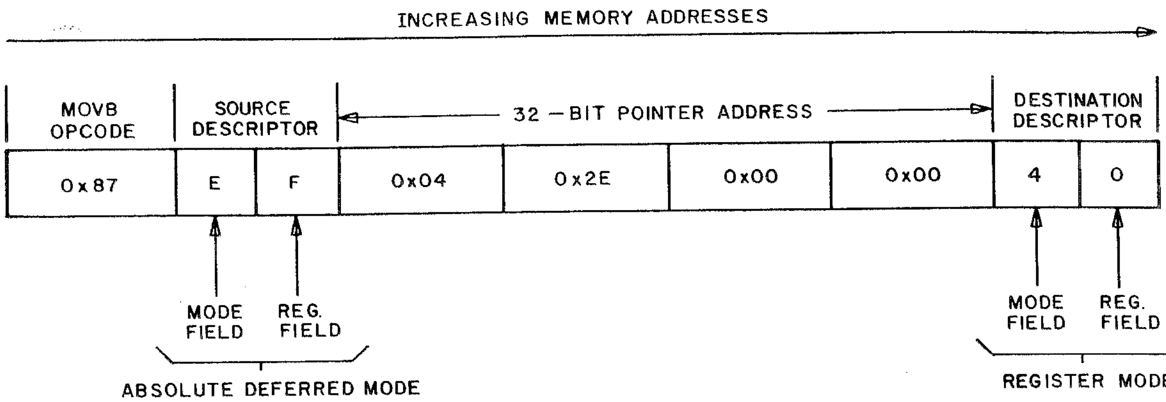

Figure 5-13. An Absolute Deferred Mode Source Operand

5.8 EXPANDED OPERAND MODE

The expanded operand mode changes the type of an operand. For example, using this mode a signed byte located in a register could be converted to an unsigned halfword stored into memory.

The expanded operand 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*}*operand*

where *operand* is an operand having any address mode except an expanded operand mode. When the expanded operand mode is used, *type* overrides the operand's normal data type, except as noted above. The new type remains in effect for the operands that follow in the instruction unless another expanded operand mode overrides it. Table 5-3 lists the syntax for *type*.

The expanded operand mode requires two descriptor bytes as shown on Figure 5-14. 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 contains the value of the new type (see Table 5-3). 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)

OxE	TYPE FIELD	MODE FIELD	REG. FIELD

Figure 5-14. Expanded Operand Mode Descriptor Bytes

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 bit (7) through 32 bits, and writes an unsigned halfword. The bytes are stored in memory as illustrated on Figure 5-15.

The expanded operand mode is illegal with coprocessor instructions and CALL, SAVE, RESTORE, SWAP INTERLOCKED, PUSHW, PUSHAW, POPW, and JSB instructions and will generate an illegal operand fault.

Table 5-3. Options for *type* in Expanded Operand Mode

Type	Syntax	Type Field (See Note)
Signed byte	**sbyte**	E7
Signed halfword	**half** or **shalf**	E6
Signed word	**word** or **sword**	E4
Unsigned byte	**byte** or **ubyte**	E3
Unsigned halfword	**uhalf**	E2
Unsigned word	**uword**	E0

Note: Type fields E1, E5, E8—E14 are reserved data types.
Type field EF is an absolute deferred data type.

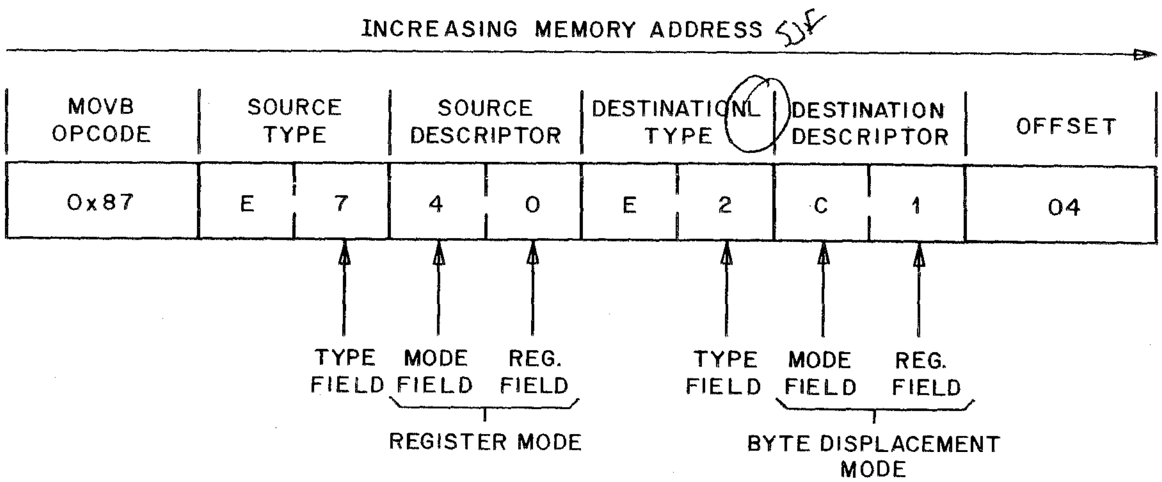

Figure 5-15. Expanded Operand Mode Example

5.9 SUMMARY

In machine language, the first byte of the operand, the *descriptor byte*, defines the operand's addressing mode. This byte consists of a mode and register field, which together define an addressing mode (the expanded operand type mode uses two descriptor bytes). Bytes following the descriptor byte contain additional data required by the addressing mode. Table 5-4 provides a summary of the addressing modes and their syntax. The descriptions within 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
%	References a processor register; use the syntax shown in Table 5-4 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).

Table 5-4. 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*(% *rn*)	12	0–10,12–15	2	–
Byte displacement deferred	**expr*(% *rn*)	13	0–10,12–15	2	–
Halfword displacement	*expr*(% *rn*)	10	0–10,12–15	3	–
Halfword displacement deferred	**expr*(% *rn*)	11	0–10,12–15	3	–
Word displacement	*expr*(% *rn*)	8	0–10,12–15	5	–
Word displacement deferred	**expr*(% *rn*)	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	% *rn*	4	0–14	1	1,3
Register deferred	(% *rn*)	5	0–10,12–14	1	1
Special Mode					
Expanded operand	{*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*.

Chapter 6
Instruction Set

CHAPTER 6. INSTRUCTION SET

CONTENTS

6. INSTRUCTION SET

The instruction set for the assembler used by the *AT&T* 3B2/3B5/3B15 Computers consists of the *WE* 32100 Microprocessor instruction set, the IS25 instructions, and the Math Acceleration Unit Instruction Set (MIS). The IS25 instruction set, discussed in more detail in **Appendix B. IS25 Instruction Set**, was designed to be machine independent and therefore it allows programs to be written for all 3B computers including earlier version 3B computers. IS25 instructions may be used in place of the *WE* 32100 Microprocessor instruction set for some applications. Since 3B2/3B5/3B15 Computers use the *WE* 32100 Microprocessor as the CPU, the discussion in this chapter is limited to the microprocessor instruction set. The remainder of this chapter will discuss the use of the microprocessor instructions and give a listing of the instructions by functional group. For a detailed listing of each instruction refer to **Appendix A. *WE* 32100 Microprocessor Instruction Set**. The MIS instructions, which are used to provide floating point support, are discussed in **10. Floating Point Support**.

6.1 *WE* 32100 MICROPROCESSOR 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 operation 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 **9. Operating System Interface**.

The microprocessor instructions are mnemonic-based assembly language statements. 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. Data types are discussed in **4.1 Data Types**.

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,opnd3,opnd4

where the mnemonic is separated from the operands by a white space (tab or space) and commas are used to separate operands. The different addressing modes and formats of the operands are discussed in 5. **Addressing Modes.**

6.1.1 Condition Flags

Bits 21 to 18 of the processor status word (PSW) contain four condition flags (N, Z, V, and C) that are affected by most instructions. The order is shown on Figure 6-1. 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 which affected 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 one. The arithmetic left shift operation sets the V bit only if a truncation error occurs. Bit, compare, and test instructions always reset V.

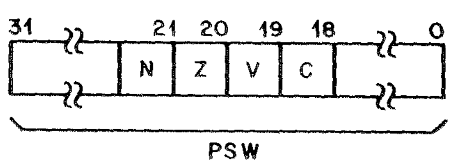

Figure 6-1. Condition Flags

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 ⩾ 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.

6.2 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 (Tables 6-1 to 6-6). Byte and cycle counts are included for the various addressing modes for each instruction. The conditions column in the instruction listing refers to the condition flag code assignment cases listed in Table 6-7.

Instruction Timing

The architecture of the *WE* 32100 CPU makes exact instruction timing calculations difficult due to the following effects:

- Addressing modes of operands
- On-chip instruction cache
- Instruction pipelining
- Instruction and data alignment
- Data dependencies

The entries in the cycle count column in Tables 9 through 15 contain the ranges, from practical best to worst case, derived from tests taking all of the above effects into consideration. It is recommended that actual benchmarks be run to more accurately measure performance. The following discussion describes the timing differences due to the above effects.

Addressing Modes of Operands. Since the instruction set is orthogonal to the addressing modes of its operands, tests were done on each applicable combination of the five basic addressing mode classes (register, absolute address, register deferred, immediate, and absolute deferred) for each instruction. Due to the nature of the addressing modes, register operations take the least time, while the absolute deferred operations take the most time to execute.

On-Chip Instruction Cache. Timing differences caused by this effect were determined by ensuring that the test instruction was in the cache prior to execution (best case) and by first flushing the cache and then executing the test instruction (worst case). By flushing the cache a prefetch had to be performed to load the cache with the instruction. Performance improvements averaged 20—60% for ALU instructions, depending on the length of the instruction, by eliminating instruction prefetches.

Instruction Pipelining. Tests to determine the timing differences due to pipelining were selected by inserting a test instruction that had potential for overlapping with two surrounding instructions and by inserting a test instruction between two branch taken instructions (using the branch instructions eliminate pipeline overlap). These tests showed on the average that pipelining saved between 2 to 6 cycles for instruction execution times.

Instruction and Data Alignment. In test runs taking this effect into account, performance increases of an average of 2 to 6 cycles were encountered for optimal alignment. Optimal alignment was obtained by placing as many of the test instruction's opcode and associated operands as possible on word boundaries. Worst case alignment minimizes alignment of the opcode and operands.

Data Dependencies. This effect was found only in four instructions: **MULW2**, **DIVW2**, **STRCPY**, and **STREND**. In the test involving the **MULW2** and **DIVW2** instructions timing is improved if at least one operand is zero. For the string instructions, the length of the string has a large impact on the instruction execution time. Since string lengths are not limited, test runs were done on strings of one byte (best case) and four bytes (worst case) to determine best and worst case timings.

6.2.1 Data Transfer Instructions

These instructions (listed in Table 6-1) 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 **6.1.1 Condition Flags**).

Table 6-1. Data Transfer Instruction Group

Instruction	Mnemonic	Opcode	Bytes	Cycles	Conditions*
Move:					
Move byte	MOVB	0x87	3–11	2–31	Case 1
Move halfword	MOVH	0x86	3–11	2–31	
Move word	MOVW	0x84	3–11	1–27	
Move address (word)	MOVAW	0x04	3–11	2–22	
Move complemented byte	MCOMB	0x8B	3–11	2–31	
Move complemented halfword	MCOMH	0x8A	3–11	2–31	
Move complemented word	MCOMW	0x88	3–11	1–27	
Move negated byte	MNEGB	0x8F	3–11	2–31	Case 2
Move negated halfword	MNEGH	0x8E	3–11	2–31	
Move negated word	MNEGW	0x8C	3–11	1–27	
Move version number	MVERNO	0x3009	2	See Note	Unchanged
Swap (Interlocked):					
Swap byte interlocked	SWAPBI	0x1F	2–6	22–33	Case 1
Swap halfword interlocked	SWAPHI	0x1E	2–6	22–33	
Swap word interlocked	SWAPWI	0x1C	2–6	18–28	
Block Operations:					
Move block of words	MOVBLW	0x3019	2	See Note	Unchanged
Field Operations:					
Extract field byte	EXTFB	0xCF	5–21	7–55	Case 1
Extract field halfword	EXTFH	0xCE	5–21	7–55	
Extract field word	EXTFW	0xCC	5–21	4–54	
Insert field byte	INSFB	0xCB	5–21	18–72	
Insert field halfword	INSFH	0xCA	5–21	18–72	
Insert field word	INSFW	0xC8	5–21	14–71	
String Operations:					
String copy	STRCPY	0x3035	2	83–182**	Unchanged
String end	STREND	0x301F	2	54–120**	

*Refer to Table 6-7 for condition flag code assignments.
**Cycle count per word access.
Note: Information Unavailable

6.2.2 Arithmetic Instructions

Arithmetic instructions (listed in Table 6-2) perform arithmetic operations on data in registers and memory. Most of these instructions have three types (specified by the last alphabetic 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 **6.1.1 Condition Flags**).

Many arithmetic operations are available as two- or three-address instructions. A two-address instruction has a source operand (*src*) and a destination operand. Three-address instructions have two source operands (*src1, src2*) and a destination operand. A few instructions also have a 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 6-2. Arithmetic Instruction Group

Instruction	Mnemonic	Opcode	Bytes	Cycles	Conditions*
Add:					
Add byte	ADDB2	0x9F	3—11	4—33	
Add halfword	ADDH2	0x9E	3—11	4—33	
Add word	ADDW2	0x9C	3—11	2—31	
Add byte, 3—address	ADDB3	0xDF	4—16	4—44	
Add halfword, 3—address	ADDH3	0xDE	4—16	4—44	
Add word, 3—address	ADDW3	0xDC	4—16	4—43	
Subtract:					
Subtract byte	SUBB2	0xBF	3—11	4—33	
Subtract halfword	SUBH2	0xBE	3—11	4—33	Case 2
Subtract word	SUBW2	0xBC	3—11	2—31	
Subtract byte, 3—address	SUBB3	0xFF	4—16	4—44	
Subtract halfword, 3—address	SUBH3	0xFE	4—16	4—43	
Subtract word, 3—address	SUBW3	0xFC	4—16	4—43	
Increment:					
Increment byte	INCB	0x93	2—6	2—24	
Increment halfword	INCH	0x92	2—6	2—24	
Increment word	INCW	0x90	2—6	1—22	
Decrement:					
Decrement byte	DECB	0x97	2—6	2—24	
Decrement halfword	DECH	0x96	2—6	2—24	
Decrement word	DECW	0x94	2—6	1—22	

*Refer to Table 6-7 for condition flag code assignments.

Table 6-2. Arithmetic Instruction Group (Continued)

Instruction	Mnemonic	Opcode	Bytes	Cycles	Conditions*
Multiply:					
Multiply byte	MULB2	0xAB	3—11	20—91	
Multiply halfword	MULH2	0xAA	3—11	20—130	Case 3
Multiply word	MULW2	0xA8	3—11	18—210	
Multiply byte, 3—address	MULB3	0xEB4—16	22—204		
Multiply halfword, 3—address	MULH3	0xEA	4—16	22—200	Case 4
Multiply word, 3—address	MULW3	0xE8	4—16	20—205	
Divide:					
Divide byte	DIVB2	0xAF	3—11	21—154	
Divide halfword	DIVH2	0xAE	3—11	21—194	Case 3
Divide word	DIVW2	0xAC	3—11	19—275	
Divide byte, 3—address	DIVB3	0xEF	4—16	23—270	
Divide halfword, 3—address	DIVH3	0xEE	4—16	23—263	Case 4
Divide word, 3—address	DIVW3	0xEC	4—16	21—275	
Modulo:					
Modulo byte	MODB2	0xA7	3—11	21—154	
Modulo halfword	MODH2	0xA6	3—11	21—194	Case 3
Modulo word	MODW2	0xA4	3—11	19—275	
Modulo byte, 3—address	MODB3	0xE7	4—16	23—270	
Modulo halfword, 3—address	MODH3	0xE6	4—16	23—263	Case 4
Modulo word, 3—address	MODW3	0xE4	21—275		
Arithmetic Shift:					
Arithmetic left shift word	ALSW3	0xC0	4—16	5—43	Case 5
Arithmetic right shift byte	ARSB3	0xC7	4—16	5—44	
Arithmetic right shift halfword	ARSH3	0xC6	4—16	5—44	Case 3
Arithmetic right shift word	ARSW3	0xC4	4—16	5—43	

*Refer to Table 6-7 for condition flag code assignments.

6.2.3 Logical Instructions

Logical instructions (listed in Table 6-3) 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 **6.1.1 Condition Flags**).

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

Table 6-3. Logical Instruction Group

Instruction	Mnemonic	Opcode	Bytes	Cycles	Conditions*
AND:					
AND byte	ANDB2	0xBB	3—11	4—33	Case 1
AND halfword	ANDH2	0xBA	3—11	4—33	
AND word	ANDW2	0xB8	3—11	2—31	
AND byte, 3—address	ANDB3	0xFB	4—16	4—44	
AND halfword, 3—address	ANDH3	0xFA	4—16	4—44	
AND word, 3—address	ANDW3	0xF8	4—16	4—43	
Exclusive OR (XOR):					
Exclusive OR byte	XORB2	0xB7	3—11	4—33	
Exclusive OR halfword	XORH2	0xB6	3—11	4—33	
Exclusive OR word	XORW2	0xB4	3—11	2—31	
Exclusive OR byte, 3—address	XORB3	0xF7	4—16	4—44	
Exclusive OR halfword, 3—address	XORH3	0xF6	4—16	4—44	
Exclusive OR word, 3—address	XORW3	0xF4	4—16	4—43	
OR:					
OR byte	ORB2	0xB3	3—11	4—33	
OR halfword	ORH2	0xB2	3—11	4—33	
OR word	ORW2	0xB0	3—11	2—31	
OR byte, 3—address	ORB3	0xF3	4—16	4—44	
OR halfword, 3—address	ORH2	0xF2	4—16	4—44	
OR word, 3—address	ORW3	0xF0	4—16	4—43	
Compare or Test:					
Compare byte	CMPB	0x3F	3—11	4—33	Case 2
Compare halfword	CMPH	0x3E	3—11	4—33	
Compare word	CMPW	0x3C	3—11	2—31	
Test byte	TSTB	0x2B	2—6	2—24	Case 6
Test halfword	TSTH	0x2A	2—6	2—24	
Test word	TSTW	0x28	2—6	1—18	
Bit test byte	BITB	0x3B	3—11	4—31	Case 1
Bit test halfword	BITH	0x3A	3—11	4—31	
Bit test word	BITW	0x38	3—11	2—30	
Clear:					
Clear byte	CLRB	0x83	2—6	2—21	Case 2
Clear halfword	CLRH	0x82	2—6	2—21	
Clear word	CLRW	0x80	2—6	1—19	
Rotate or Logical Shift:					
Rotate word	ROTW	0xD8	4—16	5—43	Case 1
Logical left shift byte	LLSB3	0xD3	4—16	5—44	
Logical left shift halfword	LLSH3	0xD2	4—16	5—44	
Logical left shift word	LLSW3	0xD0	4—16	5—43	
Logical right shift word	LRSW3	0xD4	4—16	5—43	

*Refer to Table 6-7 for condition flag code assignments.

6.2.4 Program Control Instructions

Program control instructions (listed in Table 6-4) 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 saves the address of the next instruction.

Call and return instructions for subroutines always implicitly affect the stack pointer (SP). For subroutines, call saves 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, return PC, and registers 3 through 8 (r3 — 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 the number of the six user registers r3 through r8 that will be saved and restored. A return from a function retrieves the saved pointers and registers to restore the original function's environment.

Table 6-4. Program Control Instruction Group

Instruction	Mnemonic	Opcode	Bytes	Cycles	Conditions
Unconditional Transfer:					
Branch with byte (8-bit) displacement	BRB	0x7B	2	5—16	
Branch with halfword (16-bit) displacement	BRH	0x7A	3	5—14	
Jump	JMP	0x24	2—6	7—17	
Conditional Transfers:					
Branch on carry clear byte	BCCB	0x53*	2	See Note 1	
Branch on carry clear halfword	BCCH	0x52*	3	See Note 2	
Branch on carry set byte	BCSB	0x5B*	2	See Note 1	
Branch on carry set halfword	BCSH	0x5A*	3	See Note 2	
Branch on overflow clear, byte displacement	BVCB	0x63	2	Note 1	
Branch on overflow clear, halfword displacement	BVCH	0x62	3	See Note 2	
Branch on overflow set, byte displacement	BVSB	0x6B	2	See Note 1	Unchanged
Branch on overflow set, halfword displacement	BVSH	0x6A	3	See Note 2	
Branch on equal byte (duplicate)	BEB	0x6F	2	See Note 1	
Branch on equal byte	BEB	0x7F	2	See Note 1	
Branch on equal halfword (duplicate)	BEH	0x6E	3	See Note 2	
Branch on equal halfword	BEH	0x7E	3	See Note 2	
Branch on not equal byte (duplicate)	BNEB	0x67	2	See Note 1	
Branch on not equal byte	BNEB	0x77	2	See Note 1	
Branch on not equal halfword (duplicate)	BNEH	0x66	3	See Note 2	
Branch on not equal halfword	BNEH	0x76	3	See Note 2	
Branch on less than byte (signed)	BLB	0x4B	2	See Note 1	
Branch on less than halfword (signed)	BLH	0x4A	3	See Note 2	

*Refer to Table 6-7 for condition flag code assignments.
**Indicates that opcode matches another instruction but operation is the same.
***Dependent on number of registers saved/restored.
1. 5—10 cycles during a branch not taken; 7—14 cycles during a branch taken.
2. 5—10 cycles during a branch not taken; 7—12 cycles during a branch taken.
3. 4—5 cycles during a return not taken; 13—14 cycles during a return taken.

Table 6-4. Program Control Instruction Group (Continued)

Instruction	Mnemonic	Opcode	Bytes	Cycles	Conditions
Branch on less than byte (unsigned)	BLUB	0x5B**	2	See Note 1	
Branch on less than halfword (unsigned)	BLUH	0x5A**	3	See Note 2	
Unconditional Transfer: (Continued)					
Branch on less than or equal byte (signed)	BLEB	0x4F	2	See Note 1	
Branch on less than or equal halfword (signed)	BLEH	0x4E	3	See Note 2	
Branch on less than or equal byte (unsigned)	BLEUB	0x5F	2	See Note 1	
Branch on less than or equal halfword (unsigned)	BLEUH	0x5E	3	See Note 2	
Branch on greater than byte (signed)	BGB	0x47	2	See Note 1	
Branch on greater than halfword (signed)	BGH	0x46	3	See Note 2	Unchanged
Branch on greater than byte (unsigned)	BGUB	0x57	2	See Note 1	
Branch on greater than halfword (unsigned)	BGUH	0x56	3	See Note 2	
Branch on greater than or equal byte (signed)	BGEB	0x43	2	See Note 1	
Branch on greater than or equal halfword (signed)	BGEH	0x42	3	See Note 2	
Branch on greater than or equal byte (unsigned)	BGEUB	0x53**	2	See Note 1	
Branch on greater than or equal halfword (unsigned)	BGEUH	0x52**	3	See Note 2	
Return on carry clear	RCC	0x50**	1	See Note 3	
Return on carry set	RCS	0x58**	1	See Note 3	
Return on overflow clear	RVC	0x60	1	See Note 3	
Return on overflow set	RVS	0x68	1	See Note 3	
Return on equal (unsigned)	REQLU	0x6C**	1	See Note 3	
Return on equal	REQL	0x7C**	1	See Note 3	

*Refer to Table 6-7 for condition flag code assignments.
**Indicates that opcode matches another instruction but operation is the same.
***Dependent on number of registers saved/restored.

1. 5—10 cycles during a branch not taken; 7—14 cycles during a branch taken.
2. 5—10 cycles during a branch not taken; 7—12 cycles during a branch taken.
3. 4—5 cycles during a return not taken; 13—14 cycles during a return taken.

Table 6-4. Program Control Instruction Group (Continued)

Instruction	Mnemonic	Opcode	Bytes	Cycles	Conditions
Return on not equal (unsigned)	RNEQU	0x64**	1	See Note 3	
Return on not equal	RNEQ	0x74**	1	See Note 3	
Return on less than (signed)	RLSS	0x48	1	See Note 3	Unchanged
Return on less than (unsigned)	RLSSU	0x58**	1	See Note 3	
Return on less than or equal (signed)	RLEQ	0x4C	1	See Note 3	
Return on less than or equal (unsigned)	RLEQU	0x5C	1	See Note 3	
Return on greater than (signed)	RGTR	0x44	1	See Note 3	
Return on greater than (unsigned)	RGTRU	0x54	1	See Note 3	
Return on greater than or equal (signed)	RGEQ	0x40	1	See Note 3	
Return on greater than or equal (unsigned)	RGEQU	0x50**	1	See Note 3	
Subroutine Transfer:					
Branch to subroutine, byte displacement	BSBB	0x37	2	See Note 2	
Branch to subroutine, halfword displacement	BSBH	0x36	3	See Note 2	
Jump to subroutine	JSB	0x34	2—6	7—17	
Return from subroutine	RSB	0x78	1	13—14	
Procedure Transfer:					
Save registers	SAVE	0x10	2	11—36***	
Restore registers	RESTORE	0x18	2	12—38***	
Call procedure	CALL	0x2C	7	25—36	
Return from procedure	RET	0x08	1	21—23	

*Refer to Table 6-7 for condition flag code assignments.
**Indicates that opcode matches another instruction but operation is the same.
***Dependent on number of registers saved/restored.
1. 5—10 cycles during a branch not taken; 7—14 cycles during a branch taken.
2. 5—10 cycles during a branch not taken; 7—12 cycles during a branch taken.
3. 4—5 cycles during a return not taken; 13—14 cycles during a return taken.

Procedure-call 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 6-2 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 by pushing them on the stack. It also adjusts SP and FP for each push.

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*/
```

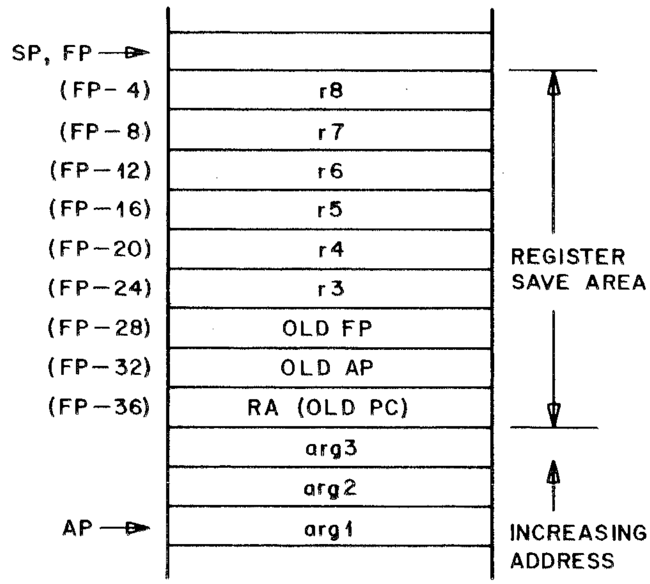

Figure 6-2. Stack After CALL-SAVE Sequence

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.

6.2.5 Coprocessor Instructions

These instructions which at present can only be used with the 3B2 Model 310 and 400 and the 3B15 Computers which contain the Math Acceleration Unit (MAU) (listed in Table 6-5), implement the interface with coprocessors. Most programmers will find it convenient to access the MAU using the MIS instruction set. 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).

6.2.6 Stack and Miscellaneous Instructions

The stack instructions (listed in Table 6-6) are used to manipulate the stack. The push and pop instructions always process a word and alter the SP. They have a source operand *src* or a 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 **as** 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 6-5. Coprocessor Instructions*

Instruction	Mnemonic	Opcode	Byte	Cycles	Conditions**
Coprocessor operation	SPOP	0x32	5	N.A.	Case 10
Coprocessor operation read single	SPOPRS	0x22	6–10	N.A.	
Coprocessor operation double	SPOPRD	0x02	6–10	N.A.	
Coprocessor operation triple	SPOPRT	0x06	6–10	N.A.	
Coprocessor operation single 2-address	SPOPS2	0x23	7–15	N.A.	
Coprocessor operation double 2-address	SPOPD2	0x03	7–15	N.A.	
Coprocessor operation triple 2-address	SPOPT2	0x07	7–15	N.A.	
Coprocessor operation write single	SPOPWS	0x33	6–10	N.A.	
Coprocessor operation write double	SPOPWD	0x13	6–10	N.A.	
Coprocessor operation write triple	SPOPWT	0x17	6–10	N.A.	

*Can only be used with 3B2 (Model 310 and 400) and 3B15 Computers.
**Refer to Table 6-7 for condition flag code assignments.
N.A. – Not available at time of production.

Table 6-6. Stack and Miscellaneous Instruction Groups

Instruction	Mnemonic	Opcode	Bytes	Cycles	Conditions*
Stack Operations:					
Push address word	PUSHAW	0xE0	2–6	9–20	Case 1
Push word	PUSHW	0xA0	2–6	8–23	
Pop word	POPW	0x20	2–6	9–23	
Miscellaneous:					Unchanged
No operation, 1 byte	NOP	0x70	1	4–11	
No operation, 2 byte	NOP2	0x73	2	4–10	
No operation, 3 byte	NOP3	0x72	3	4–10	
Breakpoint trap	BPT	0x2E	1	See Note	
Cache flush	CFLUSH	0x27	1	See Note	
Extended opcode	EXTOP	0x14	1–2	See Note	

*Refer to Table 6-7 for condition flag code assignments.
Note: Information Unavailable

Table 6-7. 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	—
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.

Table 6-7. Condition Flag Code Assignments (Continued)

Case	Condition Flags				Special Conditions*
	N(Negative)	Z(Zero)	C(Carry)	V(Overflow)	
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.

Chapter 7

Using the as Assembler

CHAPTER 7. USING THE as ASSEMBLER

CONTENTS

7. USING THE as ASSEMBLER

This chapter describes the assembler (**as**). 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 (coff).

The assembler translates operation code mnemonics and operands into the target machine bit pattern representing the particular instructions. The **as** assembler attempts to optimize the size of branch instructions, thus reducing the number of machine cycles required for a given task and improving program speed.

The assembler resolves local text labels, identifies global text symbols defined in the input files, and identifies symbols referenced but not defined.

7.1 OVERVIEW OF ASSEMBLY PROCESS

Figure 7-1 shows an overview of the assembly process. In this process the assembler source (i.e., assembly language program) is passed to the **as** assembler to create relocatable object modules. The object modules are passed to the link editor (**ld**) along with any necessary run time libraries to create an executable object module. The **ld** link editor command is described in the C Programming Language Utilities for the 3B2/3B5/3B15 Computers.

7.2 as ASSEMBLER

The assembler is called with the command line

as *options filename*

where filename ends with .s and *options* are chosen from Table 7-1.

Table 7-1. ***as* Command Line Options**

Option	Argument	Description
−G	None	Compares floating point numbers disregarding unorderedness.
−m	None	Invokes the **m4** macro processor.
−n	None	Turns off long/short address optimization.
−o	*objfile*	Places the assembled output in *objfile*.
−R	None	Removes input when done.
−U	None	Removes unreferred symbols.
−V	None	Prints the version of the assembler being run on standard error.
−Y	Directory Name	Specifies alternative directory to find m4.

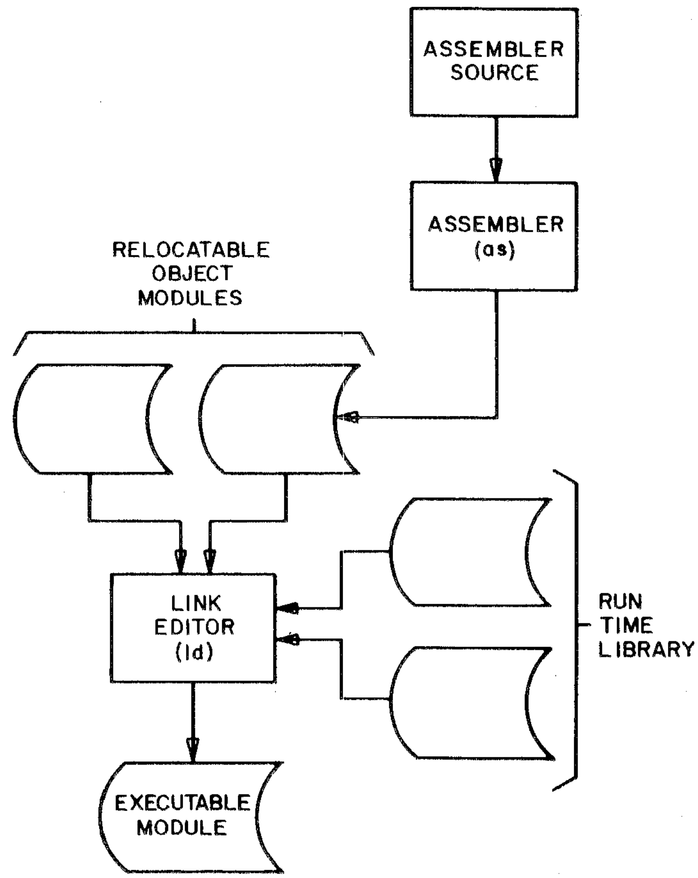

Figure 7-1. Assembly Process

The input assembly language program is read from *filename* and the output is written to an output object file. Unlike **cc**, only one file at a time may be input to **as**. If the output file name is not specified by the **−o** option, the outpuame 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 **7.3 Assembler Directives** for a more detailed description of **.align**.

7.2.1 Assembled Files

The output of the assembler is an object file. Each assembled file contains three sections: **.text**, along with optimal **.data**, and **.bss** sections. 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 executable statements, the **.data** section is used for initialized variables, and the **.bss** section is used for 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 7-2 shows these starting memory locations. Relocation of these sections is later done by the link editor (LD).

Because the assembler produces relocatable code, modular program development is possible and is encouraged.

7.2.2 Diagnostics

Errors may occur when using the assembler. The assembler outputs an error message when an error occurs. 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.

7.3 ASSEMBLER DIRECTIVES

An assembler directive is a command to the assembler that does not necessarily generate any code. Directives are distinct from executable instructions, which 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 7-2 lists all pseudo-ops alphabetically.

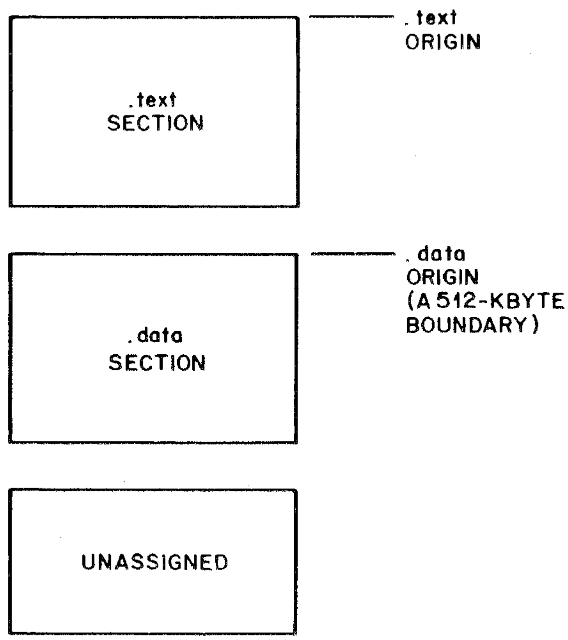

Figure 7-2. Mapping Program Sections

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**, **.word**, **.flt**, or **.double** 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.

7.3.1 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*
> **.ident** *string*

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*. The symbol *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.

.ident causes the string argument to be placed into the .comment section in the object file. The object file is a nonloaded type information section.

Table 7-2. Alphabetical List of Pseudo-Operations	
Name	**Operation**
.align *expr*	Increments the current location counter to a multiple of *expr*; *expr* must evaluate to an ABSOLUTE of 2 or 4.
.bss *sym, size, align*	Defines 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*]...	Generates initialized bytes containing the 8-bit value *val* in the current section.
.common *name,expr*	Reserves *expr* bytes of uninitialized storage for symbol name.
.data	Changes 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.
.double *val*	Generates the 64-bit floating point representation of *val*.
.endef	Ending bracket for **.def**.
.file "name"	Passes the *UNIX* System source file *name* to the assembler. Only one **.file** is allowed per assembly file.
.float *val*	Generates the 32-bit floating point representation of *val*.
.global *name*	Treats *name* as a global symbol, equivalent to storage class *extern* in the C language.
.half *val*[, *val*]...	Generates initialized halfwords containing *val* in the current section. Each *val* must be a 16-bit value.
.ident "*string*"	Places the null terminated string "*string*" in the .comment section of the output file.
.il	Indicates that a procedure has been expanded in line.
.line *expr*	Defines the source line number of the definition of block symbol "name" in **.def**. *expr* should yield an ABSOLUTE value.
.ln *line*[, *addr*]	Creates 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.
.previous	Changes the current section to the previous section. Only one level of previous section is possible.
.scl *expr*	Within **.def** give *name* the storage class of *expr*. The type of *expr* should be ABSOLUTE.

Table 7-2. Alphabetical List of Pseudo-Operations (Continued)

Name	Operation
.section *sect_name*, "*sect_type*"	Creates a section *sect_name* in the output file, of types *sect_type*, and change the current section to *sect_name*. A section type may be one or more of the following: b - bss section c - copy section i - info section d - dummy section x - executable (text) section n - noload section o - overlay section l - lib section w - data section
.set *name,expr*	Sets 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.
.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	Changes 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.
.version "*string*"	Identifies the minimum version of the assembler necessary to assemble the input file.
.word *val*[, *val*]...	Generates initialized words containing *val* in the current section. Each *val* must be a 32-bit value.
.zero *size*	Advances 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.

7.3.2 Pseudo-Operations Dealing With Symbols

The pseudo-op **.globl** is used to declare that a symbol is to be accessed by more than one object module of a single 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.

7.3.3 Assignment Pseudo-Operations

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* (see **3.4 Expressions**) 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.

Assignments are performed during the assembler's first pass over the input program. This procedure 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.

7.3.4 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

.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.

7.3.5 Alignment Pseudo-Operations

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.

7.3.6 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, while the pseudo-ops – **.flt**. and **.double** generate 32- and 64-bit floating point constants, respectively. The forms are:

.byte *expr*, ...	**.flt** *expr*, ...
.half *expr*, ...	**.double** *expr*, ...
.word *expr*, ...	

Each expression will be converted into its respective 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.

7.3.7 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 one of the values in the C leader file storeclass, h.

.type *expr* — Declares a data type for the symbol *name*. *expr* must yield a value of ABSOLUTE type that corresponds to the value of type and derived type in the header file syms,h.

.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.

7.3.8 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.

7.3.9 Line Number Pseudo-Operations

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.

7.4 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.

When the **−m** option of **as** 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 Software Generation Utilities, 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.

7.4.1 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.

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 **as**, the **−m** option can be specified on the command line. When using **cc**, 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 *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**.

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.

7.4.2 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 (cc) 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, while i remains unchanged.

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 skipping the first character, a text string has a common prefix with the string "abcdef," using the function *prefix*. 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: The address of the format string must be passed to *prefix* and that the null byte terminating the string must be explicitly coded. Also, unlike some implementations, the cc compiler does not prepend an underscore before global names. Thus, *prefix* is used in assembly code, not *_prefix*.

7.4.3 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	

Chapter 8

The dis Disassembler

CHAPTER 8. THE dis DISASSEMBLER

CONTENTS

8. THE dis DISASSEMBLER

The **dis** 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. The **dis** 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.

8.1 INVOKING THE DISASSEMBLER

To invoke the disassembler, enter the command line

dis *options files*

where *options* are chosen from Table 8-1 and *files* represents a list of object files. If no *options* are specified, all sections containing text are disassembled.

Table 8-1. ***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	None	Invokes a lookup of C source labels in the symbol table for subsequent printing.
−l	*string*	Disassembles the library file specified by *string*. For example, one would issue the command line **dis** −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.
−s	None	Disassembler in symbolic format.
−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. The **dis** 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

> **dis** *−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.

8.2 DISASSEMBLY LISTING

This section gives an sample disassembly listing and describes how it is interpreted. Three features of the **dis** listing are:

1. The disassembler prints line numbers for each C source line where a breakpoint can be set in square bracket (e.g., [5] shows the fifth source line where execution can be halted for debugging). The line numbers appear in the first column, on left hand side of the instruction corresponding to the line where a breakpoint can be inserted.
2. The disassembler, if the −s option is specified, 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 **cc** with the **−g** option and the information was not removed by a utility or link editor option).

8.2.1 Using the Disassembly Listing

(Information to be supplied.)

8.3 ERROR MESSAGES

Error messages are output when the disassembler encounters any misuse. The messages are intended to be self-explanatory.

Chapter 9
Operating System Interface

CHAPTER 9. OPERATING SYSTEM INTERFACE

CONTENTS

9. OPERATING SYSTEM INTERFACE

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.

9.1 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 9-1 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 the ***WE* 32100 Microprocessor Information Manual.**

Table 9-1. 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 9-1. 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.
- The microprocessor provides flexibility in transferring execution control between privilege levels. Control is transferred through the gate mechanism.
- 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.
- 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.

9.1.1 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.
- 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.
- A unique exception (page-write) that can be issued on any attempt to write a given page.

Detailed information or Operating System interface can be found in the ***WE* 32100 Microprocessor Information Manual.**

A: SVR3 CALLS THESE "REGIONS"

Chapter 10
Floating Point Support

CHAPTER 10. FLOATING POINT SUPPORT

CONTENTS

CHAPTER 10. FLOATING POINT SUPPORT

CONTENTS

10. FLOATING POINT SUPPORT

Support for floating point operations is provided by either the *WE* 32106 Math Acceleration Unit (MAU) Assembly Language Instruction Set (MIS) or the Floating Point Emulation (FPE) Library. Programmers using 3B2/3B5/and 3B15 Computers that contain an MAU can use either the MIS or the FPE Library (to allow programs to be compatible with all 3B2/3B5/and 3B15 Computers) when coding in assembly language, while programmers using computers which do not contain an MAU must use the FPE library. **10.1** describes the MIS and **10.2** describes the FPE library.

10.1 *WE* 32106 MATH ACCELERATION UNIT ASSEMBLY LANGUAGE INSTRUCTION SET

The following describes the *WE* 32106 Math Acceleration Unit assembly language instruction set that can be used with the 3B2/3B5/3B15 Computers. The MIS instruction set is an addition to the *WE* 32100 Microprocessor assembly language instruction set that frees the programmer from the task of generating the proper sequence of coprocessor instructions for the MAU to perform floating point operations. This section also discusses the data types used by the MIS instructions and contains an alphabetical one-page description of each MIS instruction.

10.1.1 Programmer's Overview Of *WE* 32106 Math Acceleration Unit (MAU)

This section describes the programming conventions used to support the MAU. Included in the discussion are: register usage, the immediate addressing mode notation, solutions for the problems of conditional jumps, and mixed mode arithmetic. Floating point decimal data type, which is required by the IEEE standard draft 10, is not supported by the assembler but library functions are provided to support this data type. All of the data types which are supported by the MAU (i.e., all the floating point types, words, and decimal integer) are accessible through all of the addressing modes described in **5. Addressing Modes**, except the immediate (discussed later in this section).

For a more detailed description of the MAU refer to the ***WE* 32106 Math Acceleration Unit Data Sheet**.

MAU Register Support

The MAU registers can contain only floating point data types (i.e., single, double, and double extended formats). Three out of the four MAU operand registers (numbers F0, F1, and F2) are available to the programmer. The fourth operand register (F3) is reserved for the assembler to perform substitutions for operations that use two source operands in memory. In these cases, the assembler generates two coprocessor operations, one to store one of the operands in register F3 and the second to execute the operation. The assembler does not restrict the use of F3; but since this register is reserved, the programmer is responsible for the consequences. Each MAU operand register has three names, one for each data type format:

- %s0, %s1, %s2, and %s3 for single precision
- %d0, %d1, %d2, and %d3 for double precision
- %x0, %x1, %x2, and %x3 for double extended precision.

The 0, 1, 2, and 3 correspond to the MAU operand registers F0, F1, F2, and F3, respectively.

The role of this notation is to indicate to the assembler what precision to round the register in case it is a destination and to supply *type* information in case it is a source operand. The assembler checks if the destination is narrower than the sources. If it is narrower, which is a violation of the IEEE standard draft 10, the instruction is replaced by an operation with the correct destination followed by conversion to the narrower destination as required by the IEEE standard. The fourth register is used for the intermediate result in this substitution. A default name for each register (%f0, %f1, %f2, and %f3) is also provided. The purpose of this notation is to eliminate the substitution. If it is used as a source operand, this operand does not participate in data type matching. If it is used as a destination, no substitution is done. The type for the rounding in this case is taken from the MIS instruction's opcode.

Conditional Jump Instructions

The assembler supports a set of floating point conditional jump instructions that, together with the compare instructions, supplies the predicates required by the IEEE standard draft 10. The assembler uses this set of jumps, based on the Auxiliary Status Register (ASR) flags in the MAU instead of the Processor Status Word (PSW) flags. For this purpose, the assembler reads the ASR into a word which is allocated on the stack. Although stack manipulations alter the PSW flags, corrupting the PSW flags does not affect the next floating point jump since this set uses only the ASR. The option **−G** used in the assembler command line substitutes the MIS jump instructions with the set of jump instructions which jump according to the contents of the PSW. This mode of operation of the assembler is referred to as PSW mode. If this option is taken, performance is improved and size is decreased since the access of the ASR is eliminated. However, the unordered condition is not detectable since the unordered bit of the ASR is not available in the PSW. Programmers who are willing to gain this performance, and relax the IEEE standard draft 10 requirements for jumps, can use this option.

MAU Control Bits

The MAU contains control bits in the ASR (i.e., rounding control, trap masks, and context switch control). No special instructions are implemented for these bits and the bits are controlled by reading from the ASR and writing to it. The programmer, however, must be careful to change only the desired bits in the ASR.

Immediate Operands

Although immediate operands are not supported by the MAU/CPU, the assembler provides the immediate notation for all of the supported data types. This is done by storing these operands as constants in the data section at assembly time. This static storage allocation does not involve any penalty in execution speed.

10.1.2 Data Types

The floating point data types supported by the assembler are illustrated on Figure 10-1 and are defined as:

single A 32-bit quantity that may appear at any address in memory divisible by four. Its bits are numbered from right to left starting with 0, the least significant bit (LSB), and ending with 31, the most significant bit (MSB). Bit 31 is the sign bit (s), bits 23 through 30 represent the exponential component (e) biased by 127, and bits 0 through 22 represent the fractional component (f). The value (v) of a single precision floating point number is calculated as:

a. If 0 < e < 255 then v = (−1)**s × 2**(e−127) × 1.f

b. If e = 0 and f ≠ 0 then v = (−1)**s × 2**(e−126) × 0.f

c. If e = 0 and f = 0 then v = 0

d. If e = 255 and f = 0 then v = (−1)**s × infinity

e. If e = 255 and f ≠ 0 then v = NaN

Note that NaN means "Not-a-Number" (see 10.2.3).

double A 64-bit quantity that may appear at any address in memory divisible by four. Its bits are numbered from right to left starting with 0, the LSB, and ending with 63, the MSB. Bit 63 is the sign bit (s), bits 52 through 62 represent the exponential component (e) biased by 1023, and bits 0 through 51 represent the fractional component (f). The value (v) of a double precision floating point number is calculated as:

a. If 0 < e < 2047 then v = (−1)**s × 2**(e−1023) × 1.f

b. If e = 0 and f ≠ 0 then v = (−1)**s × 2**(e−1022) × 0.f

c. If e = 0 and f = 0 then v = 0

d. If e = 2047 and f = 0 then v = (−1)**s × infinity

e. If e = 2047 and f ≠ 0 then v = NaN

double extended An 96-bit quantity that may appear at any address in memory divisible by four. Its bits are numbered from right to left starting with 0, the LSB, and ending with 95, the MSB. Bit 80 through 95 are ignored when read, and zeros are written during a write. Bit 79 is the sign bit (s), bits 64 through 78 represent the exponential component (e) biased by 16383, bit 63 represents the explicit bit (j), and bits 0 through 62 represent the fractional component (f). The value (v) of a double precision floating point number is calculated as:

a. If $0 < e < 32767$ then $v = (-1)**s \times 2**(e-16383) \times j.f$

b. If $e = 0$ and $f \neq 0$ then $v = (-1)**s \times 2**(e-16382) \times j.f$

c. If $e = 0$ and $f = 0$ then $v = 0$

d. If $e = 32767$ and $f = 0$ then $v = (-1)**s \times$ infinity

e. If $e = 32767$ and $f \neq 0$ then $v =$ NaN

Bit	31	30 23	22 0
Field	Sign	Exponent	Fraction

A. Single Data

Bit	63	62 52	51 0
Field	Sign	Exponent	Fraction

B. Double Data

Bit	95 80	79	78 64	63	62 0
Field	Unused	Sign	Exponent	J	Fraction

C. Double Extended Data

Figure 10-1. Bit Order of Data

10.1.3 MIS Instruction Listings

The following presents descriptions of each floating point instruction. The descriptions are in alphabetical order and any instruction that operates on more than one type of operand, single, double, or double extended, are listed on the same page (for quick reference to the instructions by function or mnemonic see **MIS Instructions Summary By Function** and **MIS Instructions Summary By Mnemonic** in this section). The notation used in the listings is described following.

Notation

Each instruction description contains four parts: format, operation, description, and condition indicators.

Format. 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 single, double, and double extended forms, all three forms are presented.

The syntax uses the *dst* and *src* symbols to denote operands that may be written in the address modes described in Chapter 3 of the ***WE* 32100 Microprocessor Information Manual.**

Operation. Describes the operation performed, generally, using C language syntax and the operators and symbols shown in Table 10-1.

Description. Describes the operation performed. Also, any additional explanation is included where necessary.

Result Types. Identifies the type of result that each can have upon completion of the performed operation. Table 10-2 lists the result types and their associated meanings.

Table 10-1. Assembly Language Operators and Symbols

Symbol	Description
|x|	Absolute value of 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	x less than y
x<=y	x less than or equal to y
x>y	x greater than y
x>=y	x greater than or equal to 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
address	Address of memory location
dst	Destination operand
src	Source operand
COMPARE(x,y)	Compare the contents of x and y
CONVERT(x)	Convert data type of x operation
ROUND(x)	Rounding of x operation
PC	Program counter
SQR(x)	Find square root of x operation

Table 10-2. Floating Point Result Types

Result Types	Description
FLOATING-POSITIVE-ZERO	The result of an operation is +0.
FLOATING-NEGATIVE-ZERO	The result of an operation is −0.
FLOATING-POSITIVE-NONZERO	The result of an operation has a positive sign bit and is not zero.
FLOATING-NEGATIVE-NONZERO	The result of an operation has a negative sign bit and is not zero.
FLOATING-EQUAL	In a compare operation the two operands are equal.
FLOATING-LESS	In a compare operation *src1* is less than *src2*.
FLOATING-GREATER	In a compare operation *src1* is greater than *src2*.
FLOATING-UNORDERED	In a compare operation *src1* or *src2* is a symbolic entity encoded in floating-point format and *src1* and *src2* are not equal.
NEGATIVE	The result of an operation has a negative sign bit and is not a floating-point value.
POSITIVE	The result of an operation has a positive sign bit and is not a floating-point value.
ZERO	The result of an operation is zero and is not a floating-point value.

MIS Instruction Set Descriptions

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

mfabss1
mfabsd1
mfabsx1

Floating Absolute Value One Operand

Format	mfabss1 *dst*	Single
	mfabsd1 *dst*	Double
	mfasbx1 *dst*	Double extended

Operation dst = |dst|

Description The absolute value of the contents of *dst* is taken and the floating point result is copied back into the location specified by *dst*.

Result Types
FLOATING-POSITIVE-ZERO
FLOATING-POSITIVE-NONZERO

mfabss2
mfabsd2
mfabsx2

Floating Absolute Value Two Operands

Format	mfabss2 *src,dst*	Single
	mfabsd2 *src,dst*	Double
	mfabsx2 *src,dst*	Double extended

Operation dst = |src|

Description The absolute value of the contents of *src* is taken and the floating point result is copied back into the location specified by *dst*.

Result Types
FLOATING-POSITIVE-ZERO
FLOATING-POSITIVE-NONZERO

mfadds2
mfaddd2
mfaddx2

Floating Add Two Operands

Format	mfadds2 *src,dst*	Single
	mfaddd2 *src,dst*	Double
	mfaddx2 *src,dst*	Double extended

Operation dst = dst + src

Description The contents of *src* are added to the contents of *dst*. The floating point result is copied back into the location specified by *dst*.

Result Types
FLOATING-POSITIVE-ZERO
FLOATING-NEGATIVE-ZERO
FLOATING-POSITIVE-NONZERO
FLOATING-NEGATIVE-NONZERO

mfadds3
mfaddd3
mfaddx3

Floating Add Three Operands

Format	mfadds3 *src1,src2,dst*	Single
	mfaddd3 *src1,src2,dst*	Double
	mfaddx3 *src1,src2,dst*	Double extended

Operation dst = src1 + src2

Description The contents of *src2* are added to the contents of *src1*. The floating point result is copied into the location specified by *dst*.

Result Types
FLOATING-POSITIVE-ZERO
FLOATING-NEGATIVE-ZERO
FLOATING-POSITIVE-NONZERO
FLOATING-NEGATIVE-NONZERO

mfcmps
mfcmpd
mfcompx

Floating Compare

Format

mfcmps *src1,src2* Single
mfcmpd *src1,src2* Double
mfcxmpx *src1,src2* Double extended

Operation

COMPARE(src1,src2)

Description

The contents of *src1* and *src2* are compared and appropriate condition indicators are set. This instruction is used prior to a branch or jump instruction.

Result Types

FLOATING-EQUAL
FLOATING-LESS
FLOATING-GREATER
FLOATING-UNORDERED

mfcmpts
mfcmptd
mfcmptx

Floating Compare With Trap Operation

Format	mfcmpts *src1,src2*	Single
	mfcmptd *src1,src2*	Double
	mfcmptx *src1,src2*	Double extended

Operation COMPARE(src1,src2)

Description The contents of *src1* and *src2* are compared and appropriate condition indicators are set. This instruction is used prior to a branch or jump instruction.

When an unordered condition is detected, and the invalid operation exception is enabled, an invalid operation trap occurs.

Result Types
FLOATING-EQUAL
FLOATING-LESS
FLOATING-GREATER
FLOATING-UNORDERED

mfdivs2
mfdivd2
mfdivx2

Floating Divide Two Operands

Format	mfdivs2 *src,dst*	Single
	mfdivd2 *src,dst*	Double
	mfdivx2 *src,dst*	Double extended

Operation dst = dst / src

Description The contents of *dst* are divided by the contents of *src*. The floating point result is copied back into the location specified by *dst*.

Result Types
FLOATING-POSITIVE-ZERO
FLOATING-NEGATIVE-ZERO
FLOATING-POSITIVE-NONZERO
FLOATING-NEGATIVE-NONZERO

mfdivs3
mfdivd3
mfdivx3

Floating Divide Three Operands

Format	mfdivs3 *src1,src2,dst*	Single
	mfdivd3 *src1,src2,dst*	Double
	mfdivx3 *src1,src2,dst*	Double extended

Operation dst = src2 / src1

Description The contents of *src2* are divided by the contents of *src1*. The floating point result is copied into the location specified by *dst*.

Result Types
FLOATING-POSITIVE-ZERO
FLOATING-NEGATIVE-ZERO
FLOATING-POSITIVE-NONZERO
FLOATING-NEGATIVE-NONZERO

mfmuls2
mfmuld2
mfmulx2

Floating Multiply Two Operands

Format

mfmuls2 *src,dst* Single
mfmuld2 *src,dst* Double
mfmulx2 *src,dst* Double extended

Operation

dst = dst * src

Description

The contents of *dst* are multiplied by the contents of *src*. The floating point result is copied back into the location specified by *dst*.

Result Types

FLOATING-POSITIVE-ZERO
FLOATING-NEGATIVE-ZERO
FLOATING-POSITIVE-NONZERO
FLOATING-NEGATIVE-NONZERO

mfmuls3
mfmuld3
mfmulx3

Floating Multiply Three Operands

Format	mfmuls3 *src1,src2,dst*	Single
	mfmuld3 *src1,src2,dst*	Double
	mfmulx3 *src1,src2,dst*	Double extended

Operation dst = src1 * src2

Description The contents of *src1* are multiplied by the contents of *src2*. The floating point result is copied into the location specified by *dst*.

Result Types
FLOATING-POSITIVE-ZERO
FLOATING-NEGATIVE-ZERO
FLOATING-POSITIVE-NONZERO
FLOATING-NEGATIVE-NONZERO

mfnegs1
mfnegd1
mfnegx1

Floating Negate One Operand

Format	mfnegs1 *dst*	Single
	mfnegd1 *dst*	Double
	mfnegx1 *dst*	Double extended

Operation dst = −dst

Description The two's complement value of the contents of *dst* is taken and the floating point result is copied back into the location specified by *dst*.

Result Types
FLOATING-POSITIVE-ZERO
FLOATING-NEGATIVE-ZERO
FLOATING-POSITIVE-NONZERO
FLOATING-NEGATIVE-NONZERO

mfnegs2
mfnegd2
mfnegx2

Floating Negate Two Operands

Format

mfnegs2 *src,dst* Single
mfnegd2 *src,dst* Double
mfnegx2 *src,dst* Double extended

Operation

dst = −src

Description

The negated value of the contents of *src* is taken and the floating point result is copied back into the location specified by *dst*.

Result Types

FLOATING-POSITIVE-ZERO
FLOATING-NEGATIVE-ZERO
FLOATING-POSITIVE-NONZERO
FLOATING-NEGATIVE-NONZERO

Floating Remainder Divide Two Operands

Format

mfrems2 *src,dst* Single
mfremd2 *src,dst* Double
mfremx2 *src,dst* Double extended

Operation

dst = dst % src

Description

The contents of *dst* are divided by the contents of *src*. The floating point remainder result is copied back into the location specified by *dst*.

Result Types

FLOATING-POSITIVE-ZERO
FLOATING-NEGATIVE-ZERO
FLOATING-POSITIVE-NONZERO
FLOATING-NEGATIVE-NONZERO

Floating Remainder Divide Three Operands

Format	mfrems3 *src1,src2,dst*	Single
	mfremd3 *src1,src2,dst*	Double
	mfremx3 *src1,src2,dst*	Double extended

Operation dst = src2 % src1

Description The contents of *src2* are divided by the contents of *src1*. The floating point remainder result is copied into the location specified by *dst*.

Result Types
FLOATING-POSITIVE-ZERO
FLOATING-NEGATIVE-ZERO
FLOATING-POSITIVE-NONZERO
FLOATING-NEGATIVE-NONZERO

mfrnds1
mfrndd1
mfrndx1

Floating Round to Integral Value One Operand

Format

mfrnds1 *dst* Single
mfrndd1 *dst* Double
mfrndx1 *dst* Double extended

Operation

dst = ROUND(dst)

Description

The contents of *dst* are rounded to an integral value in floating-point format and the result is copied back into the location specified by *dst*.

Result Types

FLOATING-POSITIVE-ZERO
FLOATING-NEGATIVE-ZERO
FLOATING-POSITIVE-NONZERO
FLOATING-NEGATIVE-NONZERO

mfrnds2
mfrndd2
mfrndx2

Floating Round to Integral Value Two Operands

Format	mfrnds2 *src,dst*	Single
	mfrndd2 *src,dst*	Double
	mfrndx2 *src,dst*	Double extended

Operation dst = ROUND(src)

Description The contents of *src* are rounded to an integral value and the result is copied back into the location specified by *dst*.

Result Types
FLOATING-POSITIVE-ZERO
FLOATING-NEGATIVE-ZERO
FLOATING-POSITIVE-NONZERO
FLOATING-NEGATIVE-NONZERO

mfsqrs1
mfsqrd1
mfsqrx1

Floating Square Root One Operand

Format	mfsqrs1 *dst* Single mfsqrd1 *dst* Double mfsqrx1 *dst* Double extended
Operation	dst = SQR(dst)
Description	The square root of the contents of *dst* is taken and the floating point result is copied back into the location specified by *dst*.
Result Types	FLOATING-POSITIVE-ZERO FLOATING-NEGATIVE-ZERO FLOATING-POSITIVE-NONZERO FLOATING-NEGATIVE-NONZERO

mfsqrs2
mfsqrd2
mfsqrx2

Floating Square Root Two Operands

Format	mfsqrs2 *src,dst*	Single
	mfsqrd2 *src,dst*	Double
	mfsqrx2 *src,dst*	Double extended

Operation dst = SQR(src)

Description The square root of the contents of *src* is taken and the floating point result is copied into the location specified by *dst*.

Result Types
FLOATING-POSITIVE-ZERO
FLOATING-NEGATIVE-ZERO
FLOATING-NEGATIVE-NONZERO

mfsubs2
mfsubd2
mfsubx2

Floating Subtract Two Operands

Format	mfsubs2 *src,dst*	Single
	mfsubd2 *src,dst*	Double
	mfsubx2 *src,dst*	Double extended

Operation dst = dst − src

Description The contents of *src* are subtracted from the contents of *dst*. The floating point result is copied back into the location specified by *dst*.

Result Types
FLOATING-POSITIVE-ZERO
FLOATING-NEGATIVE-ZERO
FLOATING-POSITIVE-NONZERO
FLOATING-NEGATIVE-NONZERO

mfsubs3
mfsubd3
mfsubx3

Floating Subtract Three Operands

Format	mfsubs3 *src1,src2,dst*	Single
	mfsubd3 *src1,src2,dst*	Double
	mfsubx3 *src1,src2,dst*	Double extended

Operation dst = src2 − src1

Description The contents of *src1* are subtracted from the contents of *src2*. The floating point result is copied into the location specified by *dst*.

Result Types
FLOATING-POSITIVE-ZERO
FLOATING-NEGATIVE-ZERO
FLOATING-POSITIVE-NONZERO
FLOATING-NEGATIVE-NONZERO

Floating Conditional Jumps

Format

Arithmetic Jump Operations

mjfneg *dst* Negative
mjfnz *dst* Not Zero
mjfpos *dst* Positive
mjfz *dst* Zero

Comparison Jump Operations

mjfe *dst* Equal
mjfg *dst* Greater Than
mjfge *dst* Greater Than or Equal
mjfl *dst* Less Than
mjfle *dst* Less Than or Equal
mjfne *dst* Not Equal
mjfng *dst* Not Greater Than
mjfnge *dst* Not Greater Than or Equal
mjfnl *dst* Not Less Than
mjfnle *dst* Not Less Than or Equal
mjfo *dst* Ordered
mjfu *dst* Unordered

Floating Point Exceptions

mjfde *dst* Divide-by-zero Exception
mjfexc *dst* Exception
mjfimp *dst* Imprecise
mjfio *dst* Integer Overflow
mjfioe *dst* Invalid Operation Exception
mjfoe *dst* Overflow Exception
mjfue *dst* Underflow Exception

Operation

if(condition) PC = dst

Description

When the PSW mode is not used, all jump instructions use the ASR to test the associated condition. If the condition tested is met, then the PC is replaced with the value specified by *dst*. In the PSW mode, the jump instructions use the PSW to check conditions. Also, some jump instructions are interpreted differently when assemblying in the PSW mode. These differences are noted in the following description.

Arithmetic jump operations. These operations are executed only after a floating point arithmetic operation or after one of the move or conversion operations. The conditions for these instructions are related to the result as stored in the destination. If the result is a NaN, as the result of an invalid operation, the jump is arbitrary.

Comparison jump operations. These operations are executed only after a floating point compare operation. Greater/less than means the first operand appearing in the compare instruction is greater/less than the second operand in the compare instruction.

In the PSW mode, the conditions NL, NLE, NG, and NGE are interpreted as GE, G, LE, and L respectively. The instruction, mjfu, is executed as a NOP, and mjfo is executed as an unconditional jump instruction. The rest of these instructions jump correctly if the operands are ordered but jump arbitrarily if the operands are unordered.

Floating point exceptions. When assembling a program in PSW mode, only the mjfio and mjfimp conditions are interpreted correctly, jump instructions with the other conditions are executed as NOPs.

Move Decimal Integer to Double

Format	mmov10d *src,dst*
Operation	dst = CONVERT(src)
Description	The contents of *src* are converted to a double floating point data type and that result is stored in the location specified by *dst*.
Result Types	FLOATING-POSITIVE-ZERO FLOATING-NEGATIVE-ZERO FLOATING-POSITIVE-NONZERO FLOATING-NEGATIVE-NONZERO

Move Decimal Integer to Single

Format mmov10s *src,dst*

Operation dst = CONVERT(src)

Description The contents of *src* are converted to single floating point type and that result is stored in the location specified by *dst*.

Result Types
FLOATING-POSITIVE-ZERO
FLOATING-NEGATIVE-ZERO
FLOATING-POSITIVE-NONZERO
FLOATING-NEGATIVE-NONZERO

Move Decimal Integer to Double Extended

Format mmov10x *src,dst*

Operation dst = CONVERT(src)

Description The contents of *src* are converted to a double extended floating point data type and that result is stored in the location specified by *dst.*

Result Types
FLOATING-POSITIVE-ZERO
FLOATING-NEGATIVE-ZERO
FLOATING-POSITIVE-NONZERO
FLOATING-NEGATIVE-NONZERO

Move Double to Decimal Integer

Format mmovd10 *src,dst*

Operation dst = CONVERT(src)

Description The contents of *src* are converted to decimal integer type and that result is stored in the location specified by *dst*. *dst* cannot be a MAU register.

Result Types
FLOATING-POSITIVE-ZERO
FLOATING-NEGATIVE-ZERO
FLOATING-POSITIVE-NONZERO
FLOATING-NEGATIVE-NONZERO

Move Double to Single

Format	mmovds *src,dst*
Operation	dst = CONVERT(src)
Description	The contents of *src* are converted to a single floating point data type and that result is stored in the location specified by *dst*.
Result Types	FLOATING-POSITIVE-ZERO FLOATING-NEGATIVE-ZERO FLOATING-POSITIVE-NONZERO FLOATING-NEGATIVE-NONZERO

Move Double to Binary Unsigned Word Integer

Format mmovdu *src,dst*

Operation dst = CONVERT(src)

Description The contents of *src* are converted to a unsigned word integer data type and that result is stored in the location specified by *dst*. *dst* cannot be a MAU register. This instruction cannot be used with floating-point conditional jumps.

Result Types ZERO
POSITIVE

Move Double to Binary Signed Word Integer

Format mmovdw *src,dst*

Operation dst = CONVERT(src)

Description The contents of *src* are converted to a signed word integer data type and that result is stored in the location specified by *dst*. *dst* cannot be a MAU register.

Result Types
ZERO
POSITIVE
NEGATIVE

Move Double to Double Extended

Format mmovdx *src,dst*

Operation dst = CONVERT(src)

Description The contents of *src* are converted to a double extended floating point data type and that result is stored in the location specified by *dst*.

Result Types
FLOATING-POSITIVE-ZERO
FLOATING-NEGATIVE-ZERO
FLOATING-POSITIVE-NONZERO
FLOATING-NEGATIVE-NONZERO

mmovfa
mmovfd

Move MAU Register to Memory

Format	mmovfa *address*	Move to ASR register
	mmovfd *address*	Move to data register

Operation

address = register

Description

The contents of the specified MAU register are copied into the memory location specified by *address*. The memory operand is always a word data type.

Move Single to Decimal Integer

Format mmovs10 *src,dst*

Operation dst = CONVERT(src)

Description The contents of *src* are converted to decimal integer type and that result is stored in the location specified by *dst*. *dst* cannot be a MAU register.

Result Types
FLOATING-POSITIVE-ZERO
FLOATING-NEGATIVE-ZERO
FLOATING-POSITIVE-NONZERO
FLOATING-NEGATIVE-NONZERO

Move Single to Binary Unsigned Word Integer

Format mmovsu *src,dst*

Operation dst = CONVERT(src)

Description The contents of *src* are converted to a unsigned word integer data type and that result is stored in the location specified by *dst*. *dst* cannot be a MAU register.

Result Types ZERO
POSITIVE

Move Single to Binary Signed Word Integer

Format	mmovsw *src,dst*
Operation	dst = CONVERT(src)
Description	The contents of *src* are converted to a signed word integer data type and that result is stored in the location specified by *dst*. *dst* cannot be an MAU register.
Result Types	ZERO POSITIVE NEGATIVE

Move Single to Double Extended

Format mmovsx *src,dst*

Operation dst = CONVERT(src)

Description The contents of *src* are converted to a double extended floating point data type and that result is stored in the location specified by *dst*.

Result Types
FLOATING-POSITIVE-ZERO
FLOATING-NEGATIVE-ZERO
FLOATING-POSITIVE-NONZERO
FLOATING-NEGATIVE-NONZERO

Move Memory to MAU Register

Format

mmovta *address* Move to ASR register
mmovtd *address* Move to data register

Operation

register = address

Description

The contents of the memory location specified by *address* are copied into the specified MAU register. The memory operand is always a word data type.

Move Binary Unsigned Word Integer to Double

Format mmovud *src,dst*

Operation dst = CONVERT(src)

Description The contents of *src* are converted to a double floating point data type and that result is stored in the location specified by *dst*. *src* cannot be a MAU register.

Result Types FLOATING-POSITIVE-ZERO
FLOATING-POSITIVE-NONZERO

Move Binary Unsigned Word Integer to Single

Format	mmovus *src,dst*
Operation	dst = CONVERT(src)
Description	The contents of *src* are converted to a floating single data type and that result is stored in the location specified by *dst*.
Result Types	FLOATING-POSITIVE-ZERO FLOATING-POSITIVE-NONZERO

Move Binary Unsigned Word Integer to Double Extended

Format mmovux *src,dst*

Operation dst = CONVERT(src)

Description The contents of *src* are converted to a double extended floating point data type and that result is stored in the location specified by *dst*.

Result Types
FLOATING-POSITIVE-ZERO
FLOATING-NEGATIVE-ZERO
FLOATING-POSITIVE-NONZERO
FLOATING-NEGATIVE-NONZERO

Move Binary Signed Word Integer to Double

Format mmovwd *src,dst*

Operation dst = CONVERT(src)

Description The contents of *src* are converted to a double floating point data type and that result is stored in the location specified by *dst*.

Result Types
FLOATING-POSITIVE-ZERO
FLOATING-NEGATIVE-ZERO
FLOATING-POSITIVE-NONZERO
FLOATING-NEGATIVE-NONZERO

Move Binary Signed Word Integer to Single

Format mmovws *src,dst*

Operation dst = CONVERT(src)

Description The contents of *src* are converted to a single floating point data type and that result is stored in the location specified by *dst*.

Result Types
ZERO
POSITIVE
NEGATIVE

Move Binary Signed Word Integer to Double Extended

Format	mmovwx *src,dst*
Operation	dst = CONVERT(src)
Description	The contents of *src* are converted to a double extended floating point data type and that result is stored in the location specified by *dst*.
Result Types	FLOATING-POSITIVE-ZERO FLOATING-NEGATIVE-ZERO FLOATING-POSITIVE-NONZERO FLOATING-NEGATIVE-NONZERO

Move Double Extended to Decimal Integer

Format mmovx10 *src,dst*

Operation dst = CONVERT(src)

Description The contents of *src* are converted to decimal integer type and that result is stored in the location specified by *dst*.

Result Types
FLOATING-POSITIVE-ZERO
FLOATING-NEGATIVE-ZERO
FLOATING-POSITIVE-NONZERO
FLOATING-NEGATIVE-NONZERO

Move Double Extended to Double

Format mmovxd *src,dst*

Operation dst = CONVERT(src)

Description The contents of *src* are converted to a double floating point data type and that result is stored in the location specified by *dst*.

Result Types
FLOATING-POSITIVE-ZERO
FLOATING-NEGATIVE-ZERO
FLOATING-POSITIVE-NONZERO
FLOATING-NEGATIVE-NONZERO

Move Double Extended to Single

Format mmovxs *src,dst*

Operation dst = CONVERT(src)

Description The contents of *src* are converted to a single floating point data type and that result is stored in the location specified by *dst*.

Result Types
FLOATING-POSITIVE-ZERO
FLOATING-NEGATIVE-ZERO
FLOATING-POSITIVE-NONZERO
FLOATING-NEGATIVE-NONZERO

Move Double Extended to Binary Unsigned Word Integer

Format	mmovxu *src,dst*
Operation	dst = CONVERT(src)
Description	The contents of *src* are converted to a unsigned word integer data type and that result is stored in the location specified by *dst*.
Result Types	ZERO POSITIVE

Move Double Extended to Binary Signed Word Integer

Format	mmovxw *src,dst*
Operation	dst = CONVERT(src)
Description	The contents of *src* are converted to a signed word integer data type and that result is stored in the location specified by *dst*.
Result Types	ZERO POSITIVE NEGATIVE

MIS Instructions Summary by Function

Table 10-3. MIS Instructions Summary by Function	
Mnemonic	**Name**
Data Transfer Instructions	
Move:	
mmov10d	Move decimal to double
mmov10s	Move decimal to single
mmov10x	Move decimal to double extended
mmovd10	Move double to decimal
mmovds	Move double to single
mmovdu	Move double to unsigned word
mmovdw	Move double to signed word
mmovdx	Move double to double extended
mmovs10	Move single to decimal
mmovsd	Move single to double
mmovsu	Move single to unsigned word
mmovsw	Move single to signed word
mmovsx	Move single to double extended
mmovud	Move unsigned word to double
mmovus	Move unsigned word to single
mmovux	Move unsigned word to double extended
mmovwd	Move signed word to double
mmovws	Move signed word to single
mmovwx	Move signed word to double extended
mmovx10	Move double extended to decimal
mmovxd	Move double extended to double
mmovxs	Move double extended to single
mmovxu	Move double extended to unsigned word
mmovxw	Move double extended to signed word
mmovfa	Move MAU's ASR register to memory
mmovfd	Move MAU's Data register to memory
mmovta	Move memory to MAU's ASR register
mmovtd	Move memory to MAU's Data register

Table 10-3. MIS Instructions Summary by Function (Continued)	
Mnemonic	**Name**
Arithmetic Instructions	
Add:	
mfaddd2	Add double, two operands
mfadds2	Add single, two operands
mfaddx2	Add double extended, two operands
mfaddd3	Add double, three operands
mfadds3	Add single, three operands
mfaddx3	Add double extended, three operands
Subtract:	
mfsubd2	Subtract double, two operands
mfsubs2	Subtract single, two operands
mfsubx2	Subtract double extended, two operands
mfsubd3	Subtract double, three operands
mfsubs3	Subtract single, three operands
mfsubx3	Subtract double extended, three operands
Multiply:	
mfmuld2	Multiply double, two operands
mfmuls2	Multiply single, two operands
mfmulx2	Multiply double extended, two operands
mfmuld3	Multiply double, three operands
mfmuls3	Multiply single, three operands
mfmulx3	Multiply double extended, three operands
Divide:	
mfdivd2	Divide double, two operands
mfdivs2	Divide single, two operands
mfdivx2	Divide double extended, two operands
mfdivd3	Divide double, three operands
mfdivs3	Divide single, three operands
mfdivx3	Divide double extended, three operands
Remainder Divide:	
mfremd2	Remainder divide double, two operands
mfrems2	Remainder divide single, two operands
mfremx2	Remainder divide double extended, two operands
mfremd3	Remainder divide double, three operands
mfrems3	Remainder divide single, three operands
mfremx3	Remainder divide double extended, three operands

Table 10-3. MIS Instructions Summary by Function (Continued)	
Mnemonic	**Name**
Negate:	
mfnegd1	Negate double, one operand
mfnegs1	Negate single, one operand
mfnegx1	Negate double extended, one operand
mfnegd2	Negate double, two operands
mfnegs2	Negate single, two operands
mfnegx2	Negate double extended, two operands
Round:	
mfrndd1	Round double, one operand
mfrnds1	Round single, one operand
mfrndx1	Round double extended, one operand
mfrndd2	Round double, two operands
mfrnds2	Round single, two operands
mfrndx2	Round double extended, two operands
Square Root:	
mfsqrd1	Square root double, one operand
mfsqrs1	Square root single, one operand
mfsqrx1	Square root double extended, one operand
mfsqrd2	Square root double, two operands
mfsqrs2	Square root single, two operands
mfsqrx2	Square root double extended, two operands
Logical Instructions	
Compare:	
mfcmpd	Compare double
mfcmps	Compare single
mfcmpx	Compare double extended
mfcmptd	Compare double with trap operation
mfcmpts	Compare single with trap operation
mfcmptx	Compare double extended with trap operation

Table 10-3. MIS Instructions Summary by Function (Continued)

Mnemonic	Name
Control Transfer Instructions	
mjfde	Jump if divide-by-zero exception
mjfe	Jump if equal
mjfexp	Jump if exception
mjfg	Jump if greater than
mjfge	Jump if greater than or equal
mjimp	Jump if imprecise
mjfio	Jump if integer overflow
mjfioe	Jump if invalid operation exception
mjfl	Jump if less than
mjfle	Jump if less than or equal
mjfne	Jump if not equal
mjfneg	Jump if negative
mjfng	Jump if not greater than
mjfnge	Jump if not greater than or equal
mjfnl	Jump if not less than
mjfnle	Jump if not less than or equal
mjfnz	Jump if not zero
mjfo	Jump if ordered
mjfoe	Jump if overflow exception
mjfpos	Jump if positive
mjfu	Jump if unordered
mjfue	Jump if underflow exception
mjfz	Jump if zero

MIS Instructions Summary by Mnemonic

Table 10-4. MIS Instructions Summary by Mnemonic

Mnemonic	Name
mfabsd1	Absolute value double, one operand
mfabsd2	Absolute value double, two operands
mfabss1	Absolute value single, one operand
mfabss2	Absolute value single, two operands
mfabsx1	Absolute value double extended, one operand
mfabsx2	Absolute value double extended, two operands
mfaddd2	Add double, two operands
mfaddd3	Add double, three operands
mfadds2	Add single, two operands
mfadds3	Add single, three operands
mfaddx2	Add double extended, two operands
mfaddx3	Add double extended, three operands
mfcmpd	Compare double
mfcmps	Compare single
mfcmptd	Compare double with trap operation
mfcmpts	Compare single with trap operation
mfcmptx	Compare double extended with trap operation
mfcmpx	Compare double extended
mfdivd2	Divide double, two operands
mfdivd3	Divide double, three operands
mfdivs2	Divide single, two operands
mfdivs3	Divide single, three operands
mfdivx2	Divide double extended, two operands
mfdivx3	Divide double extended, three operands
mfmuld2	Multiply double, two operands
mfmuld3	Multiply double, three operands
mfmuls2	Multiply single, two operands
mfmuls3	Multiply single, three operands
mfmulx2	Multiply double extended, two operands
mfmulx3	Multiply double extended, three operands
mfnegd1	Negate double, one operand
mfnegd2	Negate double, two operands
mfnegs1	Negate single, one operand
mfnegs2	Negate single, two operands
mfnegx1	Negate double extended, one operand
mfnegx2	Negate double extended, two operands

Table 10-4. MIS Instructions Summary by Mnemonic (Continued)

Mnemonic	Name
mfremd2	Remainder divide double, two operands
mfremd3	Remainder divide double, three operands
mfrems2	Remainder divide single, two operands
mfrems3	Remainder divide single, three operands
mfremx2	Remainder divide double extended, two operands
mfremx3	Remainder divide double extended, three operands
mfrndd1	Round double, one operand
mfrndd2	Round double, two operands
mfrnds1	Round single, one operand
mfrnds2	Round single, two operands
mfrndx1	Round double extended, one operand
mfrndx2	Round double extended, two operands
mfsqrd1	Square root double, one operand
mfsqrd2	Square root double, two operands
mfsqrs1	Square root single, one operand
mfsqrs2	Square root single, two operands
mfsqrx1	Square root double extended, one operand
mfsqrx2	Square root double extended, two operands
mfsubd2	Subtract double, two operands
mfsubd3	Subtract double, three operands
mfsubs2	Subtract single, two operands
mfsubs3	Subtract single, three operands
mfsubx2	Subtract double extended, two operands
mfsubx3	Subtract double extended, three operands
mjfde	Jump if divide-by-zero exception
mjfe	Jump if equal
mjfexp	Jump if exception
mjfg	Jump if greater than
mjfge	Jump if greater than or equal
mjfimp	Jump if imprecise
mjfio	Jump if integer overflow
mjfioe	Jump if invalid operation exception
mjfl	Jump if less than
mjfle	Jump if less than or equal
mjfne	Jump if not equal
mjfneg	Jump if negative
mjfng	Jump if not greater than
mjfnge	Jump if not greater than or equal
mjfnl	Jump if not less than

Table 10-4. MIS Instructions Summary by Mnemonic (Continued)

Mnemonic	Name
mjfnle	Jump if not less than or equal
mjfnz	Jump if not zero
mjfo	Jump if ordered
mjfoe	Jump if overflow exception
mjfpos	Jump if positive
mjfu	Jump if unordered
mjfue	Jump if underflow exception
mjfz	Jump if zero
mmov10d	Move decimal to double
mmov10s	Move decimal to single
mmov10x	Move decimal to double extended
mmovd10	Move double to decimal
mmovds	Move double to single
mmovdu	Move double to unsigned word
mmovdw	Move double to signed word
mmovdx	Move double to double extended
mmovfa	Move MAU's ASR register to memory
mmovfd	Move MAU's Data register to memory
mmovs10	Move single to decimal
mmovsd	Move single to double
mmovsu	Move single to unsigned word
mmovsw	Move single to signed word
mmovsx	Move single to double extended
mmovta	Move memory to MAU's ASR register
mmovtd	Move memory to MAU's Data register
mmovud	Move unsigned word to double
mmovus	Move unsigned word to single
mmovux	Move unsigned word to double extended
mmovwd	Move signed word to double
mmovws	Move signed word to single
mmovwx	Move signed word to double extended
mmovx10	Move double extended to decimal
mmovxd	Move double extended to double
mmovxs	Move double extended to single
mmovxu	Move double extended to unsigned word
mmovxw	Move double extended to signed word

10.2 FLOATING POINT EMULATION LIBRARY

The following describes the available floating point emulation (FPE) library that can be used with the assembler (**as**). The library must be used with 3B Computers which do not contain a *WE* 32106 Math Acceleration Unit. The FPE library can be used with computers containing a MAU, but for improved performance, the MIS instructions should be used.

The library contains a collection of functions accessible to the compiler (**cc**) or the assembler (**as**) and provides floating point primitives (e.g., addition, multiplication) and conversion between different data formats as defined by the IEEE standards. In addition, routines are provided to examine and set the rounding mode, to provide information in case of floating point exceptions, and to change the behavior of floating point exceptions.

In addition, this section discusses the data types used by the FPE library and provides descriptions of each FPE library routine in alphabetical order.

10.2.1 Assembly Language Support

Support of floating point operations for assembly language is also provided by the library libc.a. To gain access to the floating point library from an assembly language program, the programmer must first assemble the program into an object module (i.e., a .o file). Next, the programmer uses the compiler (**cc**) to automatically link the required floating point functions off libc.a as follows:

```
cc file.o
```

where file.o is the relocatable .o file which defines the integral valued function main() as the entry point of user code. This links in the required crt0.o, floating point startup, emulation routines, and if needed, the fault handling routines.

10.2.2 Data Type

The floating point data types supported by the FPE library are single and double as described in **10.1.2 Data Types**.

10.2.3 Floating Point Environment and Exception Handling

The floating point subsystem is based on the IEEE floating point standard. In this format, the largest and the smallest representable magnitudes are (as defined in the header file **values.h**):

```
#define MAXDOUBLE     1.79769313486231470e+308
#define MAXFLOAT      ((float)3.40282345538538860e+38)
#define MINDOUBLE     4.94065645841246544e−324
#define MINFLOAT      ((float)1.40129846432481707e−45)
```

Most programmers of the C language need not be concerned with the details of the floating point environment. If programmers do nothing special to handle floating point exceptions everything will work fine and if floating point traps occur (e.g., if the program tries to do a divide by zero), the program will core dump with a SIGFPE. The rest of this section discusses more details of the floating point environment and exception handling.

A new header file **ieeefp.h** has been added which defines the interface for the floating point exception and environment control. This header defines three interfaces:

- Rounding control
- Exception control
- Exception handling

The floating point arithmetic provides four rounding modes, which affect the result of most floating point operations. These modes are also defined in the header **ieeefp.h**.

```
typedef enum    fp_rnd {
   FP_RN = 0,   /* round to nearest representable number, tie -> even */
   FP_RP = 1,   /* round toward plus infinity */
   FP_RM = 2,   /* round toward minus infinity */
   FP_RZ = 3    /* round toward zero (truncate) */
} fp_rnd;
```

Programmers can check the current rounding mode with the function:

```
fp_rnd fpgetround(); /* return current rounding mode */
```

Programmers can change the rounding mode for floating point operations by the function:

```
fp_rnd fpsetround(round); /* set rounding mode, return previous */
fp_rnd round;
```

The default rounding mode is round-to-nearest. Note that in C, floating point to integer conversions are always done by truncation and the current rounding mode has no effect on these operations.

The floating point provides two kinds of special representation:

1. **Infinity**. Positive infinity in a format compares greater than all other representable numbers in the same format. Arithmetic operations on infinities are quite intuitive, e.g., adding any representable number to infinity is a valid operation, the result is positive infinity. Subtracting positive infinity from itself is invalid. If some arithmetic operation overflows and the overflow trap is disabled, in some rounding modes the delivered result is infinity.
2. **Not-a-Number (NaN)**. These floating point representations are not numbers, they may be used to carry diagnostic informations. There are two kinds of NaNs, signaling NaNs and quiet NaNs. Signaling NaNs raise the invalid operation exception whenever they are used as operands in floating point operations. Quiet NaNs

propagate through most operations without raising any exception, the result of these operation is the same quiet NaN. NaNs are sometimes produced by the arithmetic operations themselves, e.g., 0.0 divided by 0.0 when the invalid operation trap is disabled produces a quiet NaN.

Floating point operations can lead to certain exception conditions, divide by zero is a common example. There are five types of floating point exceptions:

1. **Divide by zero exception**. This exception happens when a nonzero number is divided by floating point zero.
2. **Invalid operation exception**. Operations on signaling NaNs; zero divided by zero; infinity subtracted from infinity; infinity divided by infinity; and when a quiet NaN is compared with the greater or lesser predicates, all raise invalid exceptions.
3. **Overflow exception**. This exception occurs when the result of any floating point operation is too large in magnitude to fit in the intended destination.
4. **Underflow exception**. When the underflow trap is enabled, underflow exception is signaled when a result of some operation is a tiny nonzero number smaller than the smallest representable number. When the underflow trap is disabled, underflow exception occurs only when both tinyness and loss of accuracy are detected.
5. **Inexact or imprecise exception**. This exception is signaled if the rounded result of an operation is not identical to the infinitely precise result. Inexact exceptions are quite common (e.g., 1.0/3.0 is an inexact operation). Inexact exception also occurs when the operation overflows without an overflow trap.

Note: The above examples are not an exhaustive list of the conditions when an exception can occur.

There is a sticky bit associated with each of these exceptions. Whenever any of these exceptions occur, the corresponding sticky bit is set (=1). The sticky bits are all cleared at the start of a process. After that, they are never cleared by the floating point system, just set to remember that an exception occurred.

Programmers can check the status of the sticky bits by using the function:

```
fp_except fpgetsticky(); /* return logged exceptions */
```

fp_except can have the following (not exclusive) values:

```
#define fp_except int
#define FP_X_INV 0x10 /* invalid operation exception */
#define FP_X_OFL 0x08 /* overflow exception */
#define FP_X_UFL 0x04 /* underflow exception */
#define FP_X_DZ 0x02 /* divide-by-zero exception */
#define FP_X_IMP 0x01 /* imprecise (loss of precision) exception */
```

Programmers can change the sticky bits by using the function:

```
fp_except fpsetsticky(sticky); /* change logged exceptions */
fp_except sticky;
```

There is also a trap-enable bit (mask bit) associated with each exception. When an exception occurs, if the corresponding trap bit is enabled (=1), a trap takes place. Programmers can check the status of these mask bits by using the function:

```
fp_except fpgetmask(); /* current exception mask */
```

Programmers can also selectively enable or disable any of the exceptions by calling the function:

```
fp_except fpsetmask(mask); /* set mask, return previous mask */
fp_except mask;
```

with appropriate mask values.

The default setting of the mask bits are: divide-by-zero, invalid operation, and overflow traps enabled.

The only cases where two floating point exceptions can occur together are inexact with underflow and inexact with overflow. In these cases, the trap for the inexact is taken only if the other trap is disabled.

When the trap is enabled, floating point exceptions are signaled through the standard *UNIX* System mechanism: a SIGFPE is sent to the user process. If the programmer intends to handle the trap and proceed with the program, the programmer must include the file **ieeefp.h** in at least one module of the program. The programmer can attach a handler to SIGFPE by calling the *UNIX* System signal(2) routine.

When a floating-point exception handler is entered, the global variables:

_fpftype -- floating-point fault type, and
_fpfault -- pointer to floating-point exception structure

are established. _fpftype identifies the primary exception type. Possible values for _fpftype are FP_UFLW, FP_DIVZ, INT_DIVZ etc. (see **ieeefp.h**).

_fpfault points to a structure which provides all other information about the floating point operation. The information pointed to by _fpfault includes:

1. the type of operation being performed
2. the types and values of the operands
3. the type of a trapped value (if any)
4. the desired type of the result.

The structure has the form:

```
struct fp_fault    {
    fp_op          operation;
    fp_dunion      operand[2];
    fp_dunion      t_value;
    fp_dunion      result;
```

```
};
extern struct       fp_fault *_fpfault;
```

The operation field identifies the floating point operation which raised the exception. The possible values are included in **ieeefp.h**. fp_dunion is a discriminated union which contains information about the type/format of the operands (or result) (e.g., whether the operand is in single precision or double precision). It also contains the actual values. See **ieeefp.h** for exact definitions of the different members of the union.

A user handler has the information about the floating point operation, the operands, the computed result, and the format in which the result is to be returned. The user handler can supply a result in the right format and when the handler returns, this result is used to complete the floating point operation.

10.2.4 Library Listings

The following presents descriptions of each floating point emulation function call. The descriptions are in alphabetical order and any function that operates on more than one type of operand, single or double, are listed on the same page. (For quick reference to the function calls by function or mnemonic see the following: **FPE Function Call Summary By Function** and **FPE Function Call Summary By Mnemonic**.) The notation used in the listings is described.

Notation

Each function call description contains four parts: name, synopsis, description, and exceptions.

Name. Gives the name of the function call.

Synopsis. Presents the syntax for the function call, including any required spacing and punctuation. If the function has single or double forms, both forms are presented.

Description. Describes the function performed. Also, any additional explanation is included where necessary.

Exceptions. Lists the possible exceptions which may happen. If an exception happens the associated flags in the _asr word are set.

FPE Function Call Descriptions

The FPE function calls are described in detail on the following pages. Before using these routines, the programmer must be aware of the following special cases:

1. Only the comparison calls affect the Process Status Word (PSW) bits in the *WE* 32100 Microprocessor (CPU). Other floating point calls (e.g., _fadd()) do not

set any condition codes. For the code

```
if(float_expression) statement;
```

generates a specific comparison with 0.0; i.e., the above test is treated as:

```
if(float_expression != 0.0) statement;
```

In general, for floating point operands, jump on zero/positive may not account for possible negative zeros; jump on equality does not have this problem.

2. When an exception occurs, the corresponding sticky bit is set. Additional behavior is dependent on the corresponding trap bit being set/enabled (=1) or masked/disabled (=0). A trap takes place if the mask is enabled. Note the distinction between exceptions and traps. If the trap is disabled, additional behavior depends on the operation being performed, as well as the rounding mode in effect.
3. The only cases where two exceptions can occur simultaneously are inexact with overflow and inexact with underflow. In these cases, the trap for inexact (if enabled) occurs only if the trap for overflow/underflow is disabled.
4. Signaling and quiet NaNs (Not a Number) are distinguished by the most significant bit in the explicit fraction part of the format. If this bit is zero, then the NaN is a signaling NaN, or else it is a quiet NaN.
5. When a quiet NaN has to be generated, it has a positive sign. All the fraction bits are set to one (1).
6. For format conversion of quiet NaNs (e.g., in _fdtos()), the least significant part of the fraction of the quiet NaN that fits in the fraction of the destination (less the quiet NaN bit) is placed left justified in the result.
7. If both operands are quiet NaNs, and a result is to be delivered, the resulting quiet NaN is the first argument to the function.
8. Quiet NaNs propagate through function calls without raising exceptions (except _fcmptd() and _fcmpts()).
9. The arithmetic routines never change the exception mask bits, nor clear the exception sticky bits. The programmer is provided with a set of routines to set the required masks and clear the sticky bits.
10. The float arguments to the functions and the functions returning float values are treated specially in the sense that they are not converted to double arguments/results. Normally, single precision compares and conversions are used, all other operations are performed in double precision.

NAME

_faddd – Floating Add Double
_fadds – Floating Add Single

SYNOPSIS

_faddd(src1,src2)

_fadds(src1,src2)

DESCRIPTION

Performs double/single precision addition of the double/single precision operands and returns a result in the same format.

EXCEPTIONS

Invalid-operation:

- Operations involving signaling NaN(s)
- Magnitude subtraction of infinities (e.g., +/–infinity).

Overflow

Underflow

Inexact

NAME

_fcmpd – Floating Compare Operands Double
_fcmps – Floating Compare Operands Single

SYNOPSIS

_fcmpd(src1,src2)

_fcmps(src1,src2)

DESCRIPTION

Compare the source operands. These functions are used for comparison predicates involving == and != (as well as comparison predicates like >?, if the languages are extended to include these).

The only difference between _fcmpd() and _fcmptd() is that: for _fcmpd(), quiet NaNs do not raise invalid operation exceptions. Only signaling NaNs raise invalid exceptions.

These functions return with the PSW flags in the CPU set as:

- N = 1 if src1 < src2, else 0
- Z = 1 if src1 == src2, else 0

Comparisons are always exact and never underflow or overflow.

−infinity < all finite numbers < +infinity

(+0.0 == −0.0) and (+infinity == +infinity) compare equal

Every NaN compares unordered with everything including itself.

Unordered condition raises the invalid operation exception.

EXCEPTIONS

For comparisons involving signaling NaN(s).

NAME

_fcmptd – Floating Compare With Exceptions Double
_fcmpts – Floating Compare With Exceptions Single

SYNOPSIS

_fcmptd(src1,src2)

_fcmpts(src1,src2)

DESCRIPTION

Compare with exceptions. This is the CMPE instruction of the MAU hardware. Compilers generate this call for the comparison predicates involving > and <.

These functions return with the PSW flags in the CPU set as:

- N = 1 if src1 < src2, else 0
- Z = 1 if src1 == src2, else 0

Comparisons are always exact and never underflow or overflow.

–infinity < all finite numbers < +infinity

(+0.0 == –0.0) and (+infinity == +infinity) compare equal

Every NaN compares unordered with everything including itself.

Unordered condition raises the invalid operation exception.

EXCEPTIONS

Invalid-operation: Comparison involving a signaling or quiet NaN(s) where at least one operand is a NaN.

NAME

_fdivd – Floating Divide Double
_fdivs – Floating Divide Single

SYNOPSIS

_fdivd(src1,src2)

_fdivs(src1,src2)

DESCRIPTION

Performs double/single precision division of the double/single precision operands and returns a result in the same format.

EXCEPTIONS

Divide by zero: when a nonzero number is divided by zero. If no trap occurs, the result is correctly signed infinity.

Invali

NAME
_fdtos — Convert Double to Single

SYNOPSIS
_fdtos(src)

DESCRIPTION
Convert double precision operand src to single precision format.

When a double precision quiet NaN is converted to single precision, the result contains the 22 least significant fraction bits of the source.

EXCEPTIONS
Invalid-operation: for signaling NaNs.

Overflow

Underflow

Inexact

_fltod()
_fltos()

NAME

_fltod – Convert Integer to Double
_fltos – Convert Integer to Single

SYNOPSIS

_fltod(src)

_fltos(src)

DESCRIPTION

Convert integer operand src to double/single precision floating point format.

EXCEPTIONS

Inexact for _fltos().

_fmuld()
_fmuls()

NAME

_fmuld – Floating Multiply Double
_fmuls – Floating Multiply Single

SYNOPSIS

_fmuld(src1,src2)

_fmuls(src1,src2)

DESCRIPTION

Performs double/single precision multiplication of the double/single precision operands and returns a result in the same format.

EXCEPTIONS

Invalid-operation:

- Operations involving signaling NaN(s)
- 0.0 × Infinity

Overflow

Underflow

Inexact

NAME

_fnegd – Negate Double
_fnegs – Negate Single

SYNOPSIS

_fnegd(src)

_fnegs(src)

DESCRIPTION

Negate the double/single precision operand.

_fnegd() is just src with the sign reversed, not (0.0 – src). It is treated as a non-arithmetic operation and is not checked for any exceptions.

EXCEPTIONS

None

NAME
_fstod – Convert Single to Double

SYNOPSIS
_fstod(src)

DESCRIPTION
Convert single precision operand src to double precision format.

This is one instance of the MAU operation MOVE.

When a single precision quiet NaN is converted to double precision, it contains the 22 diagnostic bits of the float source placed left justified.

EXCEPTIONS
Invalid-operation: for signaling NaNs.

NAME

_fsubd – Floating Subtract Double
_fsubs – Floating Subtract Single

SYNOPSIS

_fsubd(src1,src2)

_fsubs(src1,src2)

DESCRIPTION

Performs double/single precision subtraction of the double/single precision operands and returns a result in the same format.

EXCEPTIONS

Invalid-operation:

- Operations involving signaling NaN(s)
- Magnitude subtraction of infinities (e.g., +infinity – +infinity).

Overflow

Underflow

Inexact

NAME

_ftdtol – Convert Double to Integer
_ftstol – Convert Single to Integer

SYNOPSIS

_ftdtol(src)

_ftstol(src)

DESCRIPTION

Convert double/single precision operand src to integer format with rounding mode set to *to zero* (truncation).

Negative zero is converted to integer zero.

If integer overflow occurs, with traps disabled, the result is undefined.

If invalid operation exception occurs, and the trap is disabled, the result is undefined.

Conversion of negative floating point values to unsigned integer returns integer zero.

EXCEPTIONS

Invalid-operation: if the source operand is NaN or infinity.

Integer overflow.

NAME

_ftdtou – Convert Double to Unsigned Integer
_ftstou – Convert Single to Unsigned Integer

SYNOPSIS

_ftdtou(src)

_ftstou(src)

DESCRIPTION

Convert double/single precision operand src to unsigned integer format with rounding mode set to *to zero* (truncation).

Negative zero is converted to integer zero.

If integer overflow occurs, with traps disabled, the result is undefined.

If invalid operation exception occurs, and the trap is disabled, the result is undefined.

Conversion of negative floating point values to unsigned integer returns integer zero.

Truncation is used to convert floating point numbers to integers.

EXCEPTIONS

Invalid-operation: if the source operand is NaN or infinity.

Integer overflow.

NAME
_futod – Convert Unsigned Integer to Double
_futos – Convert Unsigned Integer to Single

SYNOPSIS
_futod(src)

_futos(src)

DESCRIPTION
Convert unsigned integer operand src to double/single precision floating point format.

EXCEPTIONS
Inexact for _futos().

isnand() isnanf()

isnand()
isnanf()

NAME

isnand — Check for NaN Double
isnanf — Check for NaN Single

SYNOPSIS

isnand(src)

isnanf(src)

DESCRIPTION

Returns true (1) if src is a NaN, or else returns false (0). Does not generate any exception, even for signaling NaNs.

EXCEPTIONS

None

FPE Function Call Summary by Function

Table 10-5. FPE Function Call Summary by Function	
Mnemonic	**Name**
Data Conversion Function Calls	
_fdtos _fstod	Convert double to single Convert single to double
_fltod _fltos	Convert integer to double Convert integer to single
_ftdtol _ftstol	Convert double to integer Convert singel to integer
_ftdtou _ftstou	Convert double to unsigned integer Convert single to unsigned integer
Arithmetic Function Calls	
Add: _faddd _fadds	 Add double Add single
Subtract: _fsubd _fsubs	 Subtract double Subtract single
Multiply: _fmuld _fmuls	 Multiply double Multiply single
Divide: _fdivd _fdivs	 Divide double Divide single
Negate: _fnegd _fnegs	 Negate double Negate single
Logical Function Calls	
Compare: _fcmpd _fcmps _fcmptd _fcmpts	 Compare double Compare single Compare with exception double Compare with exception single
Check for NaNs: isnand isnans	 Check for NaN double Check for NaN single

FPE Function Call Summary by Mnemonic

Table 10-6. FPE Function Call Summary by Mnemonic

Mnemonic	Name
_faddd	Add double
_fadds	Add single
_fcmpd	Compare double
_fcmps	Compare single
_fcmptd	Compare with exceptions double
_fcmpts	Compare with exceptions single
_fdivd	Divide double
_fdivs	Divide single
_fdtos	Convert double to single
_fltod	Convert integer to double
_fltos	Convert integer to single
_fmuld	Multiply double
_fmuls	Multiply single
_fnegd	Negate double
_fnegs	Negate single
_fstod	Convert single to double
_fsubd	Subtract double
_fsubs	Subtract single
_ftdtol	Convert double to integer
_ftstol	Convert single to integer
_ftdtou	Convert double to unsigned integer
_ftstou	Convert single to unsigned integer
_futod	Convert unsigned integer to double
_futos	Convert unsigned integer to single
isnand	Check for NaN double
isnans	Check for NaN single

Appendix A

WE 32100 Microprocessor Instruction Set

APPENDIX A. *WE* 32100 MICROPROCESSOR INSTRUCTION SET

CONTENTS

A. *WE* 32100 MICROPROCESSOR INSTRUCTION SET LISTINGS

A.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 **A.3 Instruction Set Summary by Function, A.4 Instruction Set Summary by Mnemonic**, and **A.5 Instruction Set Summary by Opcode**).

A.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 5-2: *count, dst, offset, src, width*. Program control instructions use *disp8* or *disp16* as a displacement operand. This 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 A-1.

Address Modes – Identifies the valid address modes for each operand. Refer to Table 5-4 for address mode syntax and to Table A-2 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.

A.2 INSTRUCTION SET DESCRIPTIONS

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

Table A-1. 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
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

Table A-2. 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.
3. Registers 11 and 15 may not be used in some address modes.

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

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
```

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 = *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

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

BGB
BGH

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

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
```

BGUB
BGUH

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

BLEB
BLEH

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

BLEUB
BLEUH

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

BLUB
BLUH

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

BRB
BRH

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

BSBB
BSBH

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

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

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

temp = *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).

DECB
DECH
DECW

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

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, skips 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*.

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, skips 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.

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

01	23	45	67

= increasing bits

MNEGB %r0,%r1

After: r1

FF	FF	FF	99

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 = *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

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

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
Opcodes	0x50 RCC
Operation	if (C==0) PC = *(--SP)
Address Modes	None
Condition Flags	Unchanged
Exceptions	None
Examples	RCC

RETURN ON CARRY SET

Assembler Syntax	RCS Return on carry set
Opcodes	0x58 RCS
Operation	if (C==1) PC = *(--SP)
Address Modes	None
Condition Flags	Unchanged
Exceptions	None
Examples	RCS

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
Examples	REQL

RESTORE REGISTERS

Assembler Syntax RESTORE %r*n* Restore registers

Opcodes 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.

Examples 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
Opcodes	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
Examples	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)
Opcodes	0x40 RGEQ
Operation	if ((N==0)|(Z==1)) PC = *(--SP)
Address Modes	None
Condition Flags	Unchanged
Exceptions	None
Examples	RGEQ

RETURN ON GREATER THAN OR EQUAL (UNSIGNED)

Assembler Syntax	RGEQU Return on greater than or equal (unsigned)
Opcodes	0x50 REGEQU
Operation	if (C==0) PC = *(--SP)
Address Modes	None
Condition Flags	Unchanged
Exceptions	None
Examples	RGEQU

RETURN ON GREATER THAN (SIGNED)

Assembler Syntax	RGTR Return on greater than (signed)
Opcodes	0x44 RGTR
Operation	if ((N & Z)==0) PC = *(--SP)
Address Modes	None
Condition Flags	Unchanged
Exceptions	None
Examples	RGTR

RETURN ON GREATER THAN (UNSIGNED)

Assembler Syntax	RGTRU Return on greater than
Opcodes	0x54 RGTRU
Operation	if ((C & Z)==0) PC = *(--SP)
Address Modes	None
Condition Flags	Unchanged
Exceptions	None
Examples	RGTRU

RETURN ON LESS THAN OR EQUAL (SIGNED)

Assembler Syntax	RLEQ Return on less than or equal
Opcodes	0x4C RLEQ
Operation	if ((N|Z)==1) PC = *(--SP)
Address Modes	None
Condition Flags	Unchanged
Exceptions	None
Examples	RLEQ

RETURN ON LESS THAN OR EQUAL (UNSIGNED)

Assembler Syntax	RLEQU Return on less than or equal (unsigned)
Opcodes	0x5C RLEQU
Operation	if ((C|Z)==1) PC = *(--SP)
Address Modes	None
Condition Flags	Unchanged
Exceptions	None
Examples	RLEQU

RETURN ON LESS THAN (SIGNED)

Assembler Syntax	RLSS Return on less than (signed)
Opcodes	0x48 RLSS
Operation	if ((N==1)&(Z==0)) PC = *(--SP)
Address Modes	None
Condition Flags	Unchanged
Exceptions	None
Examples	RLSS

RETURN ON LESS THAN (UNSIGNED)

Assembler Syntax RLSSU Return on less than (unsigned)

Opcodes 0x58 RLSSU

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

Address Modes None

Condition Flags Unchanged

Exceptions None

Examples RLSSU

RNEQ
RNEQU

RETURN ON NOT EQUAL

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

Opcodes
0x74 RNEQ
0x64 RNEQU

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

Address Modes None

Condition Flags Unchanged

Exceptions None

Examples RNEQ

ROTATE

Assembler Syntax	ROTW *count,src,dst* Rotate word
Opcodes	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)
Opcodes	0x78 RSB
Operation	PC = *(− −SP)
Address Modes	None
Condition Flags	Unchanged
Exceptions	None
Examples	RSB

RETURN ON OVERFLOW CLEAR

Assembler Syntax	RVC Return on overflow clear
Opcodes	0x60 RVC
Operation	if (V==0) PC = *(--SP)
Address Modes	None
Condition Flags	Unchanged
Exceptions	None
Examples	RVC

RETURN ON OVERFLOW SET

Assembler Syntax	RVS Return on overflow set
Opcodes	0x68 RVS
Operation	if (V==1) PC = *(--SP)
Address Modes	None
Condition Flags	Unchanged
Exceptions	None
Examples	RVS

SAVE REGISTERS

Assembler Syntax

SAVE %r*n* Save registers

Opcodes

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.

Examples

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 **Rèstore** 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

Opcodes
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
Determined by the coprocessor status.

Exceptions
External memory fault may occur.

Examples
SPOP 0XFFFFFFFF

Note:
Can be used only with computers containing an MAU.

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

Opcodes

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.

Examples

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

Note:

Can be used only with computers containing an MAU.

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

Opcodes

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.

Examples

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

Note: Can be used only with computers containing an MAU.

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

Opcodes

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.

Examples

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

Note: Can be used only with computers containing an MAU.

STRING COPY

Assembler Syntax STRCPY String copy

Opcodes 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,*$0x30(%r2)
SUBH2 %r0,$resulth
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

temp = 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.

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
```

EXCLUSIVE OR, 3 ADDRESS

Assembler Syntax	XORB3 *src1,src2,dst*	Exclusive OR byte, 3 address
	XORH3 *src1,src2,dst*	Exclusive OR halfword, 3 address
	XORW3 *src1,src2,dst*	Exclusive OR word, 3 address

Opcodes

0xF7 XORB3
0xF6 XORH3
0xF4 XORW3

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

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

A.3 INSTRUCTION SET SUMMARY BY FUNCTION

Table A-3. 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 A-4. 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 A-4. 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 A-5. Logical Instruction 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	0xF2
OR word, 3-address	ORW3	0xF0
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 A-6. Program Control Instruction Group

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 mnemonic with the same operation.

Table A-6. Program Control Instruction Group (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 mnemonic with the same operation.

Table A-7. 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

Note: Can be used only with computers containing a MAU.

Table A-8. 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

A.4 INSTRUCTION SET SUMMARY BY MNEMONIC

Table A-9. 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 mnemonic with the same operation.

Table A-9. 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

* Indicates that opcode matches another instruction mnemonic with the same operation.

Table A-9. Instruction Set Summary by Mnemonic (Continued)

Mnemonic	Opcode	Instruction
INCB	0x93	Increment byte
INCH	0x92	Increment halfword
INCW	0x90	Increment word
INSFB	0xCB	Insert field byte
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

Table A-9. Instruction Set Summary by Mnemonic (Continued)

Mnemonic	Opcode	Instruction
POPW	0x20	Pop word
PUSHAW	0xE0	Push address word
PUSHW	0xA0	Push word
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

* Indicates that opcode matches another instruction mnemonic with the same operation.

Table A-9. Instruction Set Summary by Mnemonic (Continued)

Mnemonic	Opcode	Instruction
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
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

A.5 Instruction Set Summary by Opcode

Table A-10. Instruction Set Summary by Opcode

Mnemonic	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
CFLUSH	0x27	Cache flush
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 A-10. 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	Ranch 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 mnemonic with the same operation.

Table A-10. 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 A-10. 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 A-10. 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
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)

Table A-10. Instruction Set Summary by Opcode (Continued)

Mnemonic	Opcode	Instruction
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
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

Table A-10. Instruction Set Summary by Opcode (Continued)

Mnemonic	Opcode	Instruction
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
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

Table A-10. Instruction Set Summary by Opcode (Continued)

Mnemonic	Opcode	Instruction
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

Appendix B

IS25 Instruction Set

APPENDIX B. IS25 INSTRUCTION SET

CONTENTS

B. IS25 INSTRUCTION SET

This appendix describes the IS25 Instruction Set. The IS25 Instruction Set was designed to be machine independent so that it could be used with all of the members of the 3B line of computers. Although, this instruction set can be used when writing assembly language programs for the 3B2/3B5/3B15 Computers it is suggested that this set of instructions be used only when necessary. Otherwise, all coding should be done in *WE* 32100 Microprocessor instructions. The remainder of this appendix describes the addressing modes available for IS25 instructions and lists each of the IS25 instructions.

B.1 ADDRESSING MODES

An addressing mode can be defined in terms of the size implied by the instruction in which it is used. The size implied by an instruction is derived either from the mnemonic operation code of the instruction (e.g., the implied size of **movb** is *byte*) or from the discussion of the semantics of the instruction (e.g., the implied size of the addressing mode for a shift count is *byte*).

An *expr* is an expression which evaluates to a value with either absolute text, data, bss type or has the external attribute at assembly time. The notation *expr* denotes the result of evaluating *expr*.

The remainder of this section discusses each of the addressing modes, summarized in Table B-1, that are used when writing code in IS25 instructions.

Table B-1. Addressing Modes For IS25 Instructions

Mode	Syntax
Absolute	
Absolute	$*expr*
Absolute deferred	*$*expr*
Displacement (from a register)	
Displacement	*expr*(%*rn*)
Displacement deferred	**expr*(%*rn*)
External Address	
External address	*expr*
External address deferred	**expr*
Immediate	
Immediate *	&*expr*
Register	
Register	%*rn*

* This mode may not be used for a destination operand.

B.1.1 Absolute Mode

Syntax: *$expr*

Effective address: value of *expr*

Operand value: data object at effective address

The value of *expr* is used as the effective address of the operand. The assembler is forced to use an absolute address for *expr*.

B.1.2 Absolute Deferred Mode

Syntax: **$expr*

Effective address: contents of word at memory location specified by *expr*

Operand value: data object at effective address

The value of *expr* is used as the address of a word in memory that contains the effective address of the operand. The assembler is forced to use an absolute address for *expr*.

B.1.3 Displacement Mode

Syntax: *expr(reg)*

Effective address: the value of the sum of the contents of *expr* and the contents of *reg*

Operand value: data object at effective address

The contents of *expr* and the contents of *reg* are added. The result is used as the effective address of the operand.

B.1.4 Displacement Deferred Mode

Syntax: **expr(reg)*

Effective address: the contents of the word at memory location specified by the sum of contents of *expr* and the contents of *reg*

Operand value: data object at effective address

The contents of *expr* and the contents of *reg* are added. The sum is used as the address of a word in memory that contains the effective address of the operand.

B.1.5 External Address Mode

Syntax: *expr*

Effective address: the contents of *expr*

Operand value: data object at effective address

The contents of *expr* is used as the effective address of the operand. The assembler chooses an appropriate addressing mode for *expr*.

B.1.6 External Address Deferred Mode

Syntax: **expr*

Effective address: contents of word at memory location specified by *expr*

Operand value: data object at effective address

The contents of *expr* is used as the address of a word in memory which contains the effective address of the operand. The assembler chooses an appropriate addressing mode for *expr*.

B.1.7 Immediate Mode

Syntax: *&expr*

Effective address: none

Operand value: contents of *expr*

The contents of *expr* is the operand. There is no effective address associated with this mode; therefore, an assembly error occurs if this mode is used as a destination or if the address is requested by the instruction. The range of values of *expr* depends on the size implied by the instruction: for bytes, 0 through (2**8)−1, halfwords, −(2**16) through (2**16)−1, and words, −(2**32) through (2**32)−1.

B.1.8 Register Mode

Syntax: **reg**

Effective address: none

Operand value: contents of *reg*

If *reg* is used as a source, the contents of *reg* are the operand. For bytes, only the lower 8 bits of *reg* are relevant; for halfwords, only the lower 16 bits of *reg* are relevant; for words, the entire contents of *reg* is relevant. If *reg* is used as a destination, the final result of the instruction is placed into *reg*. For bytes, the lower 8 bits are changed and the upper 24 bits are made zero; for halfwords, the lower 16 bits are changed and the upper 16 bits are made copies of the most significant bit of the lower 16 bits; for words, the entire 32 bits are changed. Since a register does not have an effective address, an assembly error occurs if an address is requested by the instruction.

B.2 IS25 INSTRUCTION SET LISTINGS

B.2.2 presents descriptions of each member of the IS25 instruction set.

The descriptions are in alphabetical order and any instruction that operates 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 or mnemonic see **B.2.3 Instruction Set Summary By Function** and **B.2.4 Instruction Set Summary By Mnemonic.**)

B.2.1 Notation

Each instruction description contains four parts: assembler syntax, operation, description, and result types.

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 B-1: *dst, src, count, offset, index, incr, limit, num*, and *width*.

Operation. Describes the operation performed, generally, using C language syntax and the operators and symbols shown in Table B-2.

Description. Describes the operation performed in prose. Also, any additional explanation is included where necessary.

Result Types. Identifies the type of result of the instruction that is executed.

B.2.2 IS25 Instruction Set Descriptions

The IS25 instruction set is described in detail on the following pages.

Table B-2. Assembly Language Operators and Symbols

Symbol	Description
*x	Indirection; value pointed to by x
&x	Address of 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 less than or equal to y
x>y	x greater than y
x>=y	x greater than or equal to 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
ap	Argument pointer; register 10 (r10)
BEXT(x)	Function that returns x, sign extended through 32 bits
count	Count operand
dst	Destination operand
fp	Frame pointer; register 9 (r9)
incr	Incrementer operand
index	Index operand
limit	Limit operand
num	Bit number operand
pc	Program counter; register 15 (r15)
sp	Stack pointer; register 12 (r12)
src	Source operand
tmp	Temporary storage
TRUNC(x)	Function that returns x, truncated by 1 to 3 bytes
ZEXT(x)	Function that returns x, zero extended through 32 bits

addb2
addh2
addw2

Add Two Operands

Assembler Syntax

addb2 *src,dst* Byte
addh2 *src,dst* Halfword
addw2 *src,dst* Word

Operation

dst = dst + src

Description

The contents of *src* are added to the contents of *dst*. The result is copied back into the location specified by *dst*.

Result Types

dst == 0 : Zero
dst != 0 : Non-Zero
dst > 0 : Positive
dst <= 0 : Not-Positive
dst < 0 : Negative
dst >= 0 : Not-Negative

addb3
addh3
addw3

Add Three Operands

Assembler Syntax	addb3 *src1,src2,dst*	Byte
	addh3 *src1,src2,dst*	Halfword
	addw3 *src1,src2,dst*	Word

Operation

dst = src1 + src2

Description

The contents of *src2* are added to the contents of *src1*. The result is copied into the location specified by *dst*.

Result Types

dst == 0 : Zero
dst != 0 : Non-Zero
dst > 0 : Positive
dst <= 0 : Not-Positive
dst < 0 : Negative
dst >= 0 : Not-Negative

Arithmetic Left Shift Two Operands

Assembler Syntax

alsw2 *count,dst*

Operation

dst = dst << count

Description

The contents of *dst* are shifted left the number of bits specified by *count*. The sign bit is not involved in an arithmetic left shift. Bits shifted to the left are lost *before* the sign bit. The result is stored in the location specified by *dst*.

This shift instruction operates on word destinations. *Count* is a byte operand; only the lower five bits are used (unsigned). The sign bit does not change and zeros are supplied on the right.

Result Types

dst == 0 : Zero
dst != 0 : Non-Zero
dst > 0 : Positive
dst <= 0 : Not-Positive
dst < 0 : Negative
dst >= 0 : Not-Negative

Arithmetic Left Shift Three Operands

Assembler Syntax

alsw3 *count,src,dst*

Operation

dst = src << count

Description

The contents of *src* are shifted left the number of bits specified by *count*. The sign bit is not involved in an arithmetic left shift. Bits shifted to the left are lost *before* the sign bit. The result is stored in the location specified by *dst*.

This shift instruction operates on word sources and destinations. *Count* is a byte operand; only the lower five bits are used (unsigned). The sign bit does not change and zeros are supplied on the right.

Result Types

dst == 0 : Zero
dst != 0 : Non-Zero
dst > 0 : Positive
dst <= 0 : Not-Positive
dst < 0 : Negative
dst >= 0 : Not-Negative

andb2
andh2
andw2

AND Two Operands

Assembler Syntax	andb2 *src,dst*	Byte
	andh2 *src,dst*	Halfword
	andw2 *src,dst*	Word

Operation

dst = dst & src

Description

A logical AND is performed on *dst* and *src* and the result is stored in the location specified by *dst*. The bits of each operand are ANDed on a one-to-one basis (i.e., *dst*(bit 7) & *src*(bit 7)).

Result Types

dst == 0 : Zero
dst != 0 : Non-Zero

AND Three Operands

Assembler Syntax	andb3 *src1,src2,dst*	Byte
	andh3 *src1,src2,dst*	Halfword
	andw3 *src1,src2,dst*	Word

Operation dst = src1 & src2

Description A logical AND is performed on *src1* and *src2* and the result is stored in the location specified by *dst*. The bits of each operand are ANDed on a one-to-one basis (i.e., *src1*(bit 7) & *src2(bit 7))*.

Result Types
dst == 0 : Zero
dst != 0 : Non-Zero

 arsw2

Arithmetic Right Shift Two Operands

Assembler Syntax

arsw2 *count,dst*

Operation

dst = dst >> count

Description

The contents of *dst* are shifted right the number of bits specified by *count*. The result is stored in the location specified by *dst*. The sign bit does not shift but is duplicated *count* bits to the right to make up for bits lost at the right end.

This shift instruction operates on word destinations. *Count* is a byte operand; only the lower five bits are used (unsigned).

Result Types

dst == 0 : Zero
dst != 0 : Non-Zero
dst > 0 : Positive
dst <= 0 : Not-Positive
dst < 0 : Negative
dst >= 0 : Not-Negative

Arithmetic Right Shift Three Operands

Assembler Syntax

arsw3 *count,src,dst*

Operation

dst = src >> count

Description

The contents of *src* are shifted right the number of bits specified by *count*. The result is stored in the location specified by *dst*. The sign bit does not shift but is duplicated *count* bits to the right to make up for bits lost at the right end.

This shift instruction operates on word sources and destinations. *Count* is a byte operand; only the lower five bits are used (unsigned).

Result Types

dst == 0 : Zero
dst != 0 : Non-Zero
dst > 0 : Positive
dst <= 0 : Not-Positive
dst < 0 : Negative
dst >= 0 : Not-Negative

bitb
bith
bitw

Bit Test

Assembler Syntax

bitb *src1,src2* Byte
bith *src1,src2* Halfword
bitw *src1,src2* Word

Operation

tmp = src1 & src2

Description

A logical AND is performed on the contents of *src1* and *src2*, and the result is placed in temporary storage (*tmp* is not accessible by the programmer). This instruction is used to determine if the result of a logical AND is zero or non-zero.

Result Types

tmp == 0 : Zero
tmp != 0 : Non-Zero

Call Function

Assembler Syntax

call *num,dst*

Operation

*sp = address_of_next_instruction
*(sp + 4) = ap
ap = sp − 4*num
sp = sp + 8
pc = dst

Description

The address of the next instruction is pushed onto the stack followed by the contents of the **ap**. (The contents of the **ap** are placed on the stack using the address sp+4. Note that the **sp** is *not* incremented at this point.) The **ap** register receives the value determined by subtracting (4**num*) bytes from the **sp**. This causes the **ap** register to point to the first argument of the function (remember that the function arguments were pushed onto the stack prior to calling the function). The **sp** is then incremented by 8 (two words) to point to the next available word on the stack. The 2 word increment is necessary so that the previous contents of the **ap** (placed on the stack earlier) are not overwritten. *Dst* is then stored in the **pc** causing a jump to the function.

Num is an immediate operand in the range 0 to 65535. It is the number of words of parameters to be passed to the called function.

Result Types

undefined

cmpb
cmph
cmpw

Compare

Assembler Syntax

cmpb *src1,src2* Byte
cmph *src1,src2* Halfword
cmpw *src1,src2* Word

Operation

compare src1 and src2

Description

The contents of *src1* and *src2* are compared and appropriate condition indicators are set. This instruction is used prior to a branch or jump instruction.

Since bytes are usually interpreted as unsigned quantities, the unsigned conditional jumps should be used after **cmpb**. If signed jumps are used, a byte value of 255 (which has a one in the upper bit position) is sensed as less than a byte value of 127 (which has a zero in the upper bit position).

Result Types

src1 == src2 : Equal
src1 != src2 : Not-Equal
src1 < src2 : Less
(signed comparison)
src1 <= src2 : Less-or-Equal
(signed comparison)
src1 > src2 : Greater
(singed comparison)
src1 >= src2 : Greater-or-Equal
(signed comparison)
src1 < src2 : Less-Unsigned
(unsigned comparison)
src1 <= src2 : Less-or-Equal-Unsigned
(unsigned comparison)
src1 > src2 : Greater-Unsigned
(unsigned comparison)
src1 >= src2 : Greater-or-Equal-Unsinged
(unsigned comparison)

divb2
divh2
divw2

Divide Two Operands

Assembler Syntax	divb2 *src,dst*	Byte
	divh2 *src,dst*	Halfword
	divw2 *src,dst*	Word

Operation

dst = dst / src

Description

The contents of *dst* are divided by the contents of *src*. The result is copied back into the location specified by *dst*.

Result Types

dst == 0 : Zero
dst != 0 : Non-Zero
dst > 0 : Positive
dst <= 0 : Not-Positive
dst < 0 : Negative
dst >= 0 : Not-Negative

divb3
divh3
divw3

Divide Three Operands

Assembler Syntax	divb3 *src1,src2,dst*	Byte
	divh3 *src1,src2,dst*	Halfword
	divw3 *src1,src2,dst*	Word

Operation *dst = src2 / src1*

Description The contents of *src2* are divided by the contents of *src1*. The result is copied into the location specified by *dst*.

Result Types

dst == 0 : Zero
dst != 0 : Non-Zero
dst > 0 : Positive
dst <= 0 : Not-Positive
dst < 0 : Negative
dst >= 0 : Not-Negative

Extract Field

Assembler Syntax

extzv *src,offset,width,dst*

Operation

dst = ZEXT(FIELD(offset,width,src))

Description

The field is extracted from *src* and copied into *dst*. The upper-bits of *src* are filled with zeros.

Dst is a word operand. *Offset* and *width* are immediate operands. The field is extended to 32 bits by appending high order zeros.

Result Types

undefined

Insert Field

Assembler Syntax

insv *src,offset,width,dst*

Operation

FIELD(offset,width,dst) = TRUNC(src)

Description

Src is truncated (high order bits are lost) to the same length as *width*. A copy of the truncated *src* is then inserted into *dst* with an offset of *offset*.

Src is a word operand. *Offset* and *width* are immediate operands. The high order bits of *src* are truncated in order to fit into the field.

Result Types

undefined

Conditional Jumps

Assembler Syntax	jz *dst*	Zero
	jnz *dst*	Not Zero
	jpos *dst*	Positive
	jnpos *dst*	Not Positive
	jneg *dst*	Negative
	jnneg *dst*	Not Negative
	je *dst*	Equal
	jne *dst*	Not Equal
	jl *dst*	Less Than
	jle *dst*	Less Than or Equal
	jg *dst*	Greater Than
	jge *dst*	Greater Than or Equal
	jlu *dst*	Less Than Unsigned
	jleu *dst*	Less Than or Equal Unsigned
	jgu *dst*	Greater Than Unsigned
	jgeu *dst*	Greater Than or Equal Unsigned

Operation

if(indicator_set) pc = dst

Description

If the condition indicator that a particular jump instruction tests is set, then the contents of the **pc** are replaced by contents of *dst*. Each conditional jump instruction has an optimized branch version. Branch instructions are used for displacements of 128 halfwords or less. The operation for the branch instructions are:

if(indicator_set) pc = pc + offset

Result Types

unchanged

Unconditional Jump

Assembler Syntax	jmp *dst*
Operation	pc = dst
Description	The contents of the **pc** are replaced with the contents of the st operand. This is an unconditional jump.
Result Types	unchanged

Jump to Subroutine

Assembler Syntax	**jsb** *dst*
Operation	*sp = pc sp = sp + 4 pc = dst
Description	The contents of the **pc** are saved on the stack. The **sp** is then incremented by 4 bytes (equivalent to 1 word). Finally, *dst* replaces the contents of the **pc** causing program control to continue at the subroutine at *dst*.
Result Types	unchanged

Logical Left Shift Two Operands

Assembler Syntax llsw2 *count,dst*

Operation dst = dst << count

Description The entire contents of *dst* are shifted left *count* bits. The result is stored in the location specified by *dst*. *Count* bits are lost at the right and *count* zeros are filled in at the left.

This shift instruction operates on word sources and destinations. *Count* is a byte operand; only the lower 5 bits are used (unsigned).

Result Types dst == 0 : Zero
dst != 0 : Non-Zero

Logical Left Shift Three Operands

Assembler Syntax llsw3 *count,src,dst*

Operation dst = src << count

Description The entire contents of *src* are shifted left *count* bits. The result is stored in the location specified by *dst*. *Count* bits are lost at the ~~right~~ left and *count* zeros are filled in at the ~~left~~. right

This shift instruction operates on word sources and destinations. *Count* is a byte operand; only the lower 5 bits are used (unsigned).

Result Types
dst == 0 : Zero
dst != 0 : Non-Zero

Logical Right Shift Two Operands

Assembler Syntax

lrsw3 *count,dst*

Operation

dst = dst >> count

Description

The entire contents of *dst* are shifted right *count* bits. The result is stored in the location specified by *dst*. *Count* bits are lost at the ~~left~~ right and *count* zeros are filled in at the ~~right~~ left.

This shift instruction operates on word destinations. *Count* is a byte operand; only the lower 5 bits are used (unsigned).

Result Types

dst == 0 : Zero
dst != 0 : Non-Zero

Logical Right Shift Three Operands

Assembler Syntax

lrsw3 *count,src,dst*

Operation

dst = src >> count

Description

The entire contents of *src* are shifted right *count* bits. The result is stored in the location specified by *dst*. *Count* bits are lost at the ~~left~~ right and *count* zeros are filled in at the ~~right~~ left.

This shift instruction operates on word destinations. *Count* is a byte operand; only the lower 5 bits are used (unsigned).

Result Types

dst == 0 : Zero
dst != 0 : Non-Zero

Move Complemented

Assembler Syntax	mcomb *src,dst*	Byte
	mcomh *src,dst*	Halfword
	mcomw *src,dst*	Word

Operation dst = ~ src

Description The contents of *src* are complemented (i.e., 0 bits are changed to 1 bits and 1 bits are changed to 0 bits) and the result is stored in the location specified by *dst*.

Result Types
dst == 0 : Zero
dst != 0 : Non-Zero

mnegh
mnegw

Move Negated

Assembler Syntax	mnegh *src,dst* mnegw *src,dst*	Halfword Word
Operation	dst = −src	
Description	The two's complement of the contents of *src* is copied into the location specified by *dst*. Taking the two's complement of a number negates it.	
Result Types	dst == 0 : Zero dst != 0 : Non-Zero dst > 0 : Positive dst <= 0 : Not-Positive dst < 0 : Negative dst >= 0 : Not-Negative	

modb2
modh2
modw2

Modulo Divide Two Operands

Assembler Syntax	modb2 *src,dst*	Byte
	modh2 *src,dst*	Halfword
	modw2 *src,dst*	Word

Operation

dst = dst % src

Description

The contents of *dst* are divided by the contents of *src*. If the signed result has a remainder, it is copied back into the location specified by *dst*.

Note: The percent sign (%) is the symbol for modular division.

Result Types

dst == 0 : Zero
dst != 0 : Non-Zero
dst > 0 : Positive
dst <= 0 : Not-Positive
dst < 0 : Negative
dst >= 0 : Not-Negative

Modulo Divide Three Operands

Assembler Syntax	modb3 *src1,src2,dst*	Byte
	modh3 *src1,src2,dst*	Halfword
	modw3 *src1,src2,dst*	Word

Operation dst = src2 % src1

Description The contents of *src2* are divided by the contents of *src1*. If the signed result has a remainder, it is copied into the location specified by *dst*.

Note: The percent sign (%) is the symbol for modular division.

Result Types

dst == 0 : Zero
dst != 0 : Non-Zero
dst > 0 : Positive
dst <= 0 : Not-Positive
dst < 0 : Negative
dst >= 0 : Not-Negative

Move Address

Assembler Syntax	movaw *src,dst*
Operation	dst = & src
Description	The address of *src* is copied into the location specified by *dst*. Source and destination must be word addresses if specifying memory addresses.
Result Types	dst == 0 : Zero dst != 0 : Non-Zero

movb
movh
movw

Move

Assembler Syntax	movb *src,dst*	Byte
	movb *src,dst*	Halfword
	movb *src,dst*	Word

Operation dst = src

Description The contents of *src* are copied into the location specified by *dst*.

Result Types

dst == 0 : Zero
dst != 0 : Non-Zero
dst > 0 : Positive
dst <= 0 : Not-Positive
dst < 0 : Negative
dst >= 0 : Not-Negative

movbbh
movbbw
movbhw

Move Bit Extended

Assembler Syntax

movbbh *src,dst* Byte to Halfword
movbbw *src,dst* Byte to Word
movbhw *src,dst* Halfword to Word

Operation

dst = BEXT(src)

Description

The sign bit of *src* is extended into the upper bits of *dst* by either one, two or three bytes depending on the instruction type (e.g., byte to halfword extends the sign bit one byte). The result is copied into the location specified by *dst*.

Result Types

dst == 0 : Zero
dst != 0 : Non-Zero
dst > 0 : Positive
dst <= 0 : Not-Positive
dst < 0 : Negative
dst >= 0 : Not-Negative

movblb
movblh
movblw

Move Block

Assembler Syntax

movblb	Byte
movblh	Halfword
movblw	Word

Operation

```
while(r2 > 0)
        *r1 = *r0
        r0 = r0 + implied_size
        r1 = r1 + implied_size
        r2 = r2 - 1
```

Description

Register **r0** is the starting address of the source data, register **r1** is the starting address of the destination, and register **r2** is the number items of *implied_size* to be moved. The *implied_size* is dependent on the instruction type. Values for *implied_size* can be 1 (for byte), 2 (for halfword), or 4 (for word).

After execution of the instruction, **r2** contains the value zero, **r0** contains the address of the first byte following the source of the moved data, and **r1** contains the address of the first byte following the destination of the moved data.

This instruction will not function properly if the starting address of the source block is smaller than the starting address of the destination block and the source and destination blocks overlap.

Result Types

unchanged

movthb
movtwb
movtwh

Move Truncated

Assembler Syntax

movthb *src,dst* Halfword to Byte
movtwb *src,dst* Word to Byte
movtwh *src,dst* Word to Halfword

Operation

dst = TRUNC(src)

Description

The uppermost bits of the contents of *src* are truncated by the amount indicated by the instruction type (i.e., halfword to byte - high order byte is lost, word to byte - upper 3 bytes are lost, and word to halfword - upper 2 bytes are lost). The result is copied into the location specified by *dst*. For condition indicator settings, the result of a halfword destination is interpreted as a 16 bit signed 2's complement number; the result of a byte destination is interpreted as an 8 bit unsigned binary number. If *dst* is a register, a move truncated halfword to byte and a move truncated word to byte put the byte result into bits 7—0 of the register and puts zeros into bits 31—8 (zero extension); a move truncated word to halfword puts the halfword result into bits 15—0 of the register and copies bit 15 into bits 31—16 (sign extension).

Result Types

dst == 0 : Zero
dst != 0 : Non-Zero
dst > 0 : Positive
dst <= 0 : Not-Positive
dst < 0 : Negative
dst >= 0 : Not-Negative

movzbh
movzbw
movzhw

Move Zero Extended

Assembler Syntax	movzbh *src,dst*	Byte to Halfword
	movzbw *src,dst*	Byte to Word
	movzhw *src,dst*	Halfword to Word

Operation dst = ZEXT(src) upper bits made zero

Description The contents of *src* are expanded to the same size as *dst*. Extended bits are set to zero. The result is copied into the location specified by *dst*.

Result Types
dst == 0 : Zero
dst != 0 : Non-Zero

Multiply Two Operands

Assembler Syntax	mulb2 *src,dst*	Byte
	mulh2 *src,dst*	Halfword
	mulw2 *src,dst*	Word

Operation

dst = dst * src

Description

The contents of *dst* are multiplied by the contents of *src*. The result is copied back into the location specified by *dst*.

Result Types

dst == 0 : Zero
dst != 0 : Non-Zero
dst > 0 : Positive
dst <= 0 : Not-Positive
dst < 0 : Negative
dst >= 0 : Not-Negative

Multiply Three Operands

Assembler Syntax	mulb3 *src1,src2,dst*	Byte
	mulh3 *src1,src2,dst*	Halfword
	mulw3 *src1,src2,dst*	Word

Operation dst = src1 * src2

Description The contents of *src1* are multiplied by the contents of *src2*. The result is copied into the location specified by *dst*.

Result Types
dst == 0 : Zero
dst != 0 : Non-Zero
dst > 0 : Positive
dst <= 0 : Not-Positive
dst < 0 : Negative
dst >= 0 : Not-Negative

orb2
orh2
orw2

Or Two Operands

Assembler Syntax

orb2 *src,dst* Byte
orh2 *src,dst* Halfword
orw2 *src,dst* Word

Operation

dst = dst | src

Description

A logical OR is performed on *dst* and *src* and the result is stored in the location specified by *dst*. The bits of each operand are ORed on a one-to-one basis (i.e., *dst*(bit 7) | *src*(bit 7)).

Result Types

dst == 0 : Zero
dst != 0 : Non-Zero

orb3
orh3
orw3

UPPER CASE

Or Three Operands

Assembler Syntax	orb3 *src1,src2,dst*	Byte
	orh3 *src1,src2,dst*	Halfword
	orw3 *src1,src2,dst*	Word

Operation dst = src1 | src2

Description A logical OR is performed on *src1* and *src2* and the result is stored in the location specified by *dst*. The bits of each operand are ORed on a one-to-one basis (i.e., *src1* (bit 7) | *src2(bit 7))*.

Result Types
dst == 0 : Zero
dst != 0 : Non-Zero

Push Address

Assembler Syntax

pushaw *src*

Operation

tmp = src
*sp = tmp
sp = sp + 4

Description

The contents of *src* are placed in *tmp* (temporary storage). The contents of *tmp* are then placed on the stack and the **sp** is incremented.

Source must be a word address if specifying a memory address.

Result Types

dst == 0 : Zero
dst != 0 : Non-Zero

Push Bit Extended

Assembler Syntax

pushbb *src* Byte
pushbh *src* Halfword

Operation

tmp = BEXT(src)
*sp = tmp
sp = sp + 4

Description

The high order bit of *src* is extended into the high order two or three bytes of *tmp* (temporary storage) depending on the instruction type. The low order byte or halfword of *src* is copied into the low order byte or halfword of *tmp*. *Tmp* is pushed onto the stack and the **sp** is incremented.

Result Types

dst == 0 : Zero
dst != 0 : Non-Zero
dst > 0 : Positive
dst <= 0 : Not-Positive
dst < 0 : Negative
dst >= 0 : Not-Negative

Push

Assembler Syntax

pushw *src*

Operation

tmp = src
*sp = tmp
sp = sp + 4

Description

The contents of *src* are placed in *tmp* (temporary storage). The contents of *tmp* are then pushed onto the stack and the **sp** is incremented.

Result Types

dst == 0 : Zero
dst != 0 : Non-Zero
dst > 0 : Positive
dst <= 0 : Not-Positive
dst < 0 : Negative
dst >= 0 : Not-Negative

Push Zero Extended

Assembler Syntax	pushzb *src* pushzh *src*	Byte Halfword

Operation

tmp = ZEXT(src)
*sp = tmp
sp = sp + 4

Description

The high order two or three bytes of the *src* are filled with zeros and then copied into *tmp* (temporary storage). *Tmp* is then pushed onto the stack and the **sp** is incremented.

Result Types

dst == 0 : Zero
dst != 0 : Non-Zero

Return from Function

Assembler Syntax

ret *num*

Operation

tmp = ap
restore from save area num registers beginning with **r8** and counting downward; adjust **sp** to contain address of saved **pc**
fp = *(sp + 8)
ap = *(sp + 4)
pc = *sp
sp = tmp

Description

After the contents of the **ap** are placed in *tmp* (temporary storage), all automatic variables and save area registers are popped off the stack.

Num is an immediate operand in the range 0 to 6. It specifies which registers are to be restored (e.g., if *num* is &3, registers **r8, r7**, and **r6** will be restored). The effect of this instruction are undefined if either:

- **fp** is not the same as it was after the execution of the save instruction that created the function activation on top of the stack.
- *num* is not the same as it was after the execution of the save instruction that created the function activation on top of the stack.

Result Types

undefined

Return from Subroutine

Assembler Syntax	rsb
Operation	sp = sp - 4 pc = *sp
Description	This instruction is usually the last instruction in a subroutine. Before using this instruction any values placed on the stack by the subroutine must have been removed. When this instruction is executed, the top word on the stack is copied into the **pc**. This is the address of the instruction following the **jsb** instruction which called the subroutine.
Result Types	unchanged

Save Registers

Assembler Syntax
save *num*

Operation
*sp = fp
store in save area num registers beginning with **r8** and counting downward; adjust **sp** to contain address of first word above save area
fp = sp

Description
This instruction should be the first instruction in a subroutine. The main purpose of this instruction is to save the contents of some general purpose registers before the subroutine changes any of their contents. The registers that can be saved are: **r3** through **r8**. The **save** instruction can also be used to allocate up to 15 words on the stack.

Num is an immediate operand in the range 0 to 6. It is the number of registers to save (e.g., if *num* is &2, registers **r8** and **r7** are saved). Note that registers **r0, r1,** and **r2** cannot be saved.

Result Types
undefined

subb2
subh2
subw2

Subtract Two Operands

Assembler Syntax	subb2 *src,dst*	Byte
	subh2 *src,dst*	Halfword
	subw2 *src,dst*	Word

Operation dst = dst - src

Description The contents of *src* are subtracted from the contents of *dst*. The result is copied back into the location specified by *dst*.

Result Types
dst == 0 : Zero
dst != 0 : Non-Zero
dst > 0 : Positive
dst <= 0 : Not-Positive
dst < 0 : Negative
dst >= 0 : Not-Negative

subb3
subh3
subw3

Subtract Three Operands

Assembler Syntax	subb3 *src1,src2,dst*	Byte
	subh3 *src1,src2,dst*	Halfword
	subw3 *src1,src2,dst*	Word

Operation

dst = src1 - src2

Description

The contents of *src1* are subtracted from the contents of *src2*. The result is copied into the location specified by *dst*.

Result Types

dst == 0 : Zero
dst != 0 : Non-Zero
dst > 0 : Positive
dst <= 0 : Not-Positive
dst < 0 : Negative
dst >= 0 : Not-Negative

udivb2
usivh2
udivw2

Unsigned Divide Two Operands

Assembler Syntax	udivb2 *src,dst* Byte udivh2 *src,dst* Halfword udivw2 *src,dst* Word
Operation	dst = dst / src
Description	The contents of *dst* are divided by the contents of *src*. The unsigned result is copied back into the location specified by *dst*.
Result Types	dst == 0 : Zero dst != 0 : Non-Zero dst > 0 : Positive dst <= 0 : Not-Positive dst < 0 : Negative dst >= 0 : Not-Negative

Unsigned Divide Three Operands

Assembler Syntax	udivb3 *src1,src2,dst*	Byte
	udivh3 *src1,src2,dst*	Halfword
	udivw3 *src1,src2,dst*	Word

Operation dst = src2 / src1

Description The contents of *src2* are divided by the contents of *src1*. The unsigned result is copied into the location specified by *dst*.

Result Types

dst == 0 : Zero
dst != 0 : Non-Zero
dst > 0 : Positive
dst <= 0 : Not-Positive
dst < 0 : Negative
dst >= 0 : Not-Negative

umodb2
umodh2
umodw2

Unsigned Modulo Divide Two Operands

Assembler Syntax

umodb2 *src,dst* Byte
umodh2 *src,dst* Halfword
umodw2 *src,dst* Word

Operation

dst = dst % src

Description

The contents of *dst* are divided by the contents of *src*. If the unsigned result has a remainder, it is copied back into the location specified by *dst*.

Note: The percent sign (%) is the symbol for modular division.

Result Types

dst == 0 : Zero
dst != 0 : Non-Zero
dst > 0 : Positive
dst <= 0 : Not-Positive
dst < 0 : Negative
dst >= 0 : Not-Negative

Unsigned Modulo Divide Three Operands

Assembler Syntax	umodb3 *src1,src2,dst*	Byte
	umodh3 *src1,src2,dst*	Halfword
	umodw3 *src1,src2,dst*	Word

Operation

dst = src2 % src1

Description

The contents of *src2* are divided by the contents of *src1*. If the unsigned result has a remainder, it is copied into the location specified by *dst*.

Note: The percent sign (%) is the symbol for modular division.

Result Types

dst == 0 : Zero
dst != 0 : Non-Zero
dst > 0 : Positive
dst <= 0 : Not-Positive
dst < 0 : Negative
dst >= 0 : Not-Negative

umulb2
umulh2
umulw2

Unsigned Multiply Two Operands

Assembler Syntax	umul2 *src,dst* Byte umulh2 *src,dst* Halfword umulw2 *src,dst* Word
Operation	dst = dst * src
Description	The unsigned contents of *dst* and *src* are multiplied and the result is copied back into the location specified by *dst*.
Result Types	dst == 0 : Zero dst != 0 : Non-Zero dst > 0 : Positive dst <= 0 : Not-Positive dst < 0 : Negative dst >= 0 : Not-Negative

umulb3
umulh3
umulw3

Unsigned Multiply Three Operands

Assembler Syntax

umul3 *src1,src2,dst* Byte
umulh3 *src1,src2,dst* Halfword
umulw3 *src1,src2,dst* Word

Operation

dst = src1 * src2

Description

The unsigned contents of *src1* and *src2* are multiplied and the result is copied into the location specified by *dst.*

Result Types

dst == 0 : Zero
dst != 0 : Non-Zero
dst > 0 : Positive
dst <= 0 : Not-Positive
dst < 0 : Negative
dst >= 0 : Not-Negative

xorb2
xorh2
xorw2

Exclusive OR Two Operands

Assembler Syntax	xorb2 *src,dst*	Byte
	xorh2 *src,dst*	Halfword
	xorw2 *src,dst*	Word

Operation dst = dst ^src

Description A logical XOR (exclusive OR) is performed on *dst* and *src* and the result is stored in the location specified by *dst*. The bits of each operand are XORed on a one-to-one basis (i.e., *dst*(bit 7) XOR *src*(bit 7)).

Result Types
dst == 0 : Zero
dst != 0 : Non-Zero

Exclusive OR Three Operands

Assembler Syntax	xorb3 *src1,src2,dst*	Byte
	xorh3 *src1,src2,dst*	Halfword
	xorw3 *src1,src2,dst*	Word
Operation	dst = src1^src2	
Description	A logical XOR (exclusive OR) is performed on *src1* and *src2* and the result is stored in the location specified by *dst*. The bits of each operand are XORed on a one-to-one basis (i.e., *src1*(bit 7) XOR *src2*(bit 7)).	
Result Types	dst == 0 : Zero dst != 0 : Non-Zero	

B.2.3 IS25 Instruction Set Summary by Function

Table B-3. Data Transfer Instructions

Mnemonic	Name
Move:	
mcomb	Move complemented byte
mcomh	Move complemented halfword
mcomw	Move complemented word
mnegh	Move negated halfword
mnegw	Move negated word
movaw	Move address (word)
movb	Move byte
movh	Move halfword
movw	Move word
movbbh	Move bit extended byte to halfword
movbbw	Move bit extended byte to word
movbhw	Move bit extended halfword to word
movthb	Move truncated halfword to byte
movtwb	Move truncated word to byte
movtwh	Move truncated word to halfword
movzbh	Move zero extended byte to halfword
movzbw	Move zero extended byte to word
movzhw	Move zero extended halfword to word
Block Operations:	
movbl b	Move block byte
movblh	Move block halfword
movblw	Move block word
Field Operations:	
extzv	Extract field
insv	Insert field

Table B-4. Arithmetic Instructions

Mnemonic	Name
Add:	
addb2	Add byte, two operands
addh2	Add halfword, two operands
addw2	Add word, two operands
addb3	Add byte, three operands
addh3	Add halfword, three operands
addw3	Add word, three operands
Subtract:	
subb2	Subtract byte, two operands
subh2	Subtract halfword, two operands
subw2	Subtract word, two operands
subb3	Subtract byte, three operands
subh3	Subtract halfword, three operands
subw3	Subtract word, three operands
Multiply:	
mulw2	Multiply word, two operands
mulw3	Multiply word, three operands
umulw2	Unsigned multiply word, two operands
umulw3	Unsigned multiply word, three operands
Divide:	
divw2	Divide word, two operands
divw3	Divide word, three operands
udivw2	Unsigned Divide word, two operands
udivw3	Unsigned Divide word, three operands
Modulo:	
modw2	Modulo word, two operands
modw3	Modulo word, three operands
umodw2	Unsigned modulo word, two operands
umodw3	Unsigned modulo word, three operands
Arithmetic Shifts:	
alsw2	Arithmetic left shift word, two operands
alw3 alsw3	Arithmetic left shift word, three operands
arsw2	Arithmetic right shift word, two operands
arsw3	Arithmetic right shift word, three operands

Table B-5. Logical Instructions

Mnemonic	Name
AND:	
andb2	AND byte, two operands
andh2	AND halfword, two operands
andw2	AND word, two operands
andb3	AND byte, three operands
andh3	AND halfword, three operands
andw3	AND word, three operands
OR:	
orb2	OR byte, two operands
orh2	OR halfword, two operands
orw2	OR word, two operands
orb3	OR byte, three operands
orh3	OR halfword, three operands
orw3	OR word, three operands
Exclusive OR:	
xorb2	Exclusive OR byte, two operands
xorh2	Exclusive OR halfword, two operands
xorw2	Exclusive OR word, two operands
xorb2	Exclusive OR byte, three operands
xorh2	Exclusive OR halfword, three operands
xorw2	Exclusive OR word, three operands
Compare or Test:	
cmpb	Compare byte
cmph	Compare halfword
cmpw	Compare word
bitb	Bit test byte
bith	Bit test halfword
bitw	Bit test word
Logical Shifts:	
llsw2	Logical left shift word, two operands
llsw3	Logical left shift word, three operands
lrsw2	Logical right shift word, two operands
lrsw3	Logical right shift word, three operands

Table B-6. Program Control Instructions

Mnemonic	Name
Unconditional Transfer:	
jmp	Jump
Conditional Transfers:	
je	Jump equal
jne	Jump not equal
jg	Jump greater
jge	Jump greater or equal
jgu	Jump greater unsigned
jgeu	Jump greater or equal unsigned
jl	Jump less
jlu	Jump less unsigned
jle	Jump less or equal
jleu	Jump less or equal unsigned
jneg	Jump negative
jnneg	Jump not negative
jpos	Jump positive
jnpos	Jump not positive
jz	Jump zero
jnz	Jump not zero
Subroutine Transfer:	
jsb	Jump to subroutine
rsb	Return from subroutine
Procedure Transfer:	
call	Call procedure
ret	Return from procedure
save	Save registers

Table B-7. Stack Instructions

Mnemonic	Name
pushaw	Push address (word)
pushbb	Push extended byte
pushbh	Push extended halfword
pushw	Push word
pushzb	Push zero extended byte
pushzh	Push zero extended halfword

B.2.4 IS25 Instruction Set Summary by Mnemonic

Table B-8. IS25 Instruction Set Summary by Mnemonic

Mnemonic	Name
addb2	Add byte, two operands
addb3	Add byte, three operands
addh2	Add halfword, two operands
addh3	Add halfword, three operands
addw2	Add word, two operands
addw3	Add word, three operands
alsw2	Arithmetic left shift word, two operands
alw3 alsw3	Arithmetic left shift word, three operands
andb2	AND byte, two operands
andb3	AND byte, three operands
andh2	AND halfword, two operands
andh3	AND halfword, three operands
andw2	AND word, two operands
andw3	AND word, three operands
arsw2	Arithmetic right shift word, two operands
arsw3	Arithmetic right shift word, three operands
bitb	Bit test byte
bith	Bit test halfword
bitw	Bit test word
call	Call procedure
cmpb	Compare byte
cmph	Compare halfword
cmpw	Compare word
divw2	Divide word, two operands
divw3	Divide word, three operands
extzv	Extract field
insv	Insert field
je	Jump equal
jg	Jump greater
jge	Jump greater or equal
jgeu	Jump greater or equal unsigned

Table B-8. IS25 Instruction Set Summary by Mnemonic (Continued)

Mnemonic	Name
jgu	Jump greater unsigned
jl	Jump less
jle	Jump less or equal
jleu	Jump less or equal unsigned
jlu	Jump less unsigned
jmp	Jump
jne	Jump not equal
jneg	Jump negative
jnneg	Jump not negative
jnpos	Jump not positive
jnz	Jump not zero
jpos	Jump positive
jsb	Jump to subroutine
jz	Jump zero
llsw2	Logical left shift word, two operands
llsw3	Logical left shift word, three operands
lrsw2	Logical right shift word, two operands
lrsw3	Logical right shift word, three operands
mcomb	Move complemented byte
mcomh	Move complemented halfword
mcomw	Move complemented word
mnegh	Move negated halfword
mnegw	Move negated word
modw2	Modulo word, two operands
modw3	Modulo word, three operands
movaw	Move address (word)
movb	Move byte
movbbh	Move bit extended byte to halfword
movbbw	Move bit extended byte to word
movbhw	Move bit extended halfword to word
movblb	Move block byte
movblh	Move block halfword
movblw	Move block word
movh	Move halfword
movthb	Move truncated halfword to byte
movtwb	Move truncated word to byte
movtwh	Move truncated word to halfword
movw	Move word
movzbh	Move zero extended byte to halfword
movzbw	Move zero extended byte to word

Table B-8. IS25 Instruction Set Summary by Mnemonic (Continued)

Mnemonic	Name
movzhw	Move zero extended halfword to word
mulw2	Multiply word, two operands
mulw3	Multiply word, three operands
orb2	OR byte, two operands
orb3	OR byte, three operands
orh2	OR halfword, two operands
orh3	OR halfword, three operands
orw2	OR word, two operands
orw3	OR word, three operands
pushaw	Push address (word)
pushbb	Push extended byte
pushbh	Push extended halfword
pushw	Push word
pushzb	Push zero extended byte
pushzh	Push zero extended halfword
ret	Return from procedure
rsb	Return from subroutine
save	Save registers
subb2	Subtract byte, two operands
subb3	Subtract byte, three operands
subh2	Subtract halfword, two operands
subh3	Subtract halfword, three operands
subw2	Subtract word, two operands
subw3	Subtract word, three operands
udivw2	Unsigned Divide word, two operands
udivw3	Unsigned Divide word, three operands
umodw2	Unsigned modulo word, two operands
umodw3	Unsigned modulo word, three operands
umulw2	Unsigned multiply word, two operands
umulw3	Unsigned multiply word, three operands
xorb2	Exclusive OR byte, two operands
xorb3	Exclusive OR byte, three operands
xorh2	Exclusive OR halfword, two operands
xorh3	Exclusive OR halfword, three operands
xorw2	Exclusive OR word, two operands
xorw3	Exclusive OR word, three operands

Appendix C

Sample Programs

C. SAMPLE PROGRAMS

(Not Available at Time of Publication)

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.

Addressing mode – A method of forming the effective memory address of an operand(s) in an instruction. Examples of addressing modes include register, register displacement, immediate, and absolute deferred addressing.

Alignment – The assignment of instructions and data to specific addresses, i.e., word boundaries, to increase system performance.

Architecture – Breakdown of CPU structure into various units and registers.

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.

Assembler directive – A special command to the assembler which is generally not translated into machine code. Directives allow the programmer to set starting addresses of instructions and data, and to initialize variables, for example. Assembler directives are also referred to as pseudo-operations.

Assembly language – A programming language consisting primarily of mnemonics and symbolic addresses. Assembly language statements are translated by an assembler program to corresponding machine language instructions.

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.

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.

Central Processing Unit (CPU) – The portion of a computer which includes the logic to control the interpretation and execution of machine instructions, the arithmetic and logic unit, and various registers for data storage and addressing. A microcomputer's CPU is usually a single chip called a microprocessor.

Comment – Statements inserted in a program for documentation purposes. Comments are ignored by the assembler or compiler.

Complementary metal oxide semiconductor (CMOS) – a fabrication technology using complementary N-channel and P-channel MOS field effect transistors to provide low power dissipation and high noise immunity.

Condition Code (NZVC) – The flags in this 4-bit field reflect the resulting status of the most recent instruction execution that affects them. The four flags are negative (N), zero (Z), overflow (V), and carry (C).

Condition flags – Single bits denoting the result of an operation performed by the computer. Examples are negative, zero, and carry bits.

Coprocessor – A support processor that operates synchronously with the CPU to provide greater throughput in arithmetic or I/O functions.

Descriptor byte — An 8-bit quantity defining an operand's addressing mode and register fields.

Disassembler — A utility program which produces an assembly language listing from machine code.

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.

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.

Expression — A sequence of operands separated by operators.

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 functions local variables.

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.

High level language — A programming language consisting of statements which represent procedures rather than individual machine instructions. High level language statements are usually translated by a compiler program into a series of machine language instructions. Examples of high level languages are FORTRAN, BASIC, PASCAL, and C-language.

Interrupt — A means by which external devices may request service by the microprocessor.

Interrupt stack pointer (ISP) — User register that contains the 32-bit memory address of the top of the interrupt stack.

Label — A symbolic name used in a program to identify the location of an instruction or data.

Machine language — A programming language in which each instruction is specified by numerical values. Machine language programs can be loaded directly into memory and executed.

Macro — A sequence of instructions referenced by a name. A macro processor replaces the name by the sequence. Macros enhance programming languages by making them readable or by tailoring them to specific applications.

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 instructions.

Math Acceleration Unit (MAU) — A coprocessor providing floating point arithmetic capability for the *WE* 32100 Microprocessor.

Mnemonic — Symbolic names or abbreviations of assembly language instructions which denote the operation performed.

Negate — To drive a signal to its inactive state.

Operand – Data on which an operation is performed by an instruction.

Operating system – Software controlling the overall operation of a computer. Controls memory allocation, input and output operations, and job scheduling.

Pipelining – Overlapping the execution of instructions to increase the microprocessor's performance.

Pointer – a register or memory location containing an address.

Processor 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.

Processor status work (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.

Pseudo operation – See Assembler directive.

Register – A CPU storage unit holding bits or words.

Register deferred mode – An address mode that uses a register name, embedded in an operand, that 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, that contains data to be used by the instruction.

Stack – A reserved area of memory where the CPU saves return addresses and register data. The stack is a last-in-first-out (LIFO) queue that supports efficient subroutine linkage and local variable storage.

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 on) the stack or the last item retrieved (popped) from the stack.

Symbol – A name recognized by an assembler and used as a label, mnemonic, or operand.

Wait-state – Idle periods that may be generated during a bus transaction to allow slow peripherals to handshake with the microprocessor.

Word – A 32-bit quantity that may appear at any register divisibly by 4.

3-state – To place an input in a high-impedance state.

as – Assembler

ASR – Auxiliary status register

AP – Argument pointer

BSS – Bounded static storage

C – Condition flag bit carry

CAD – Computer-aided design

CMOS – Complimentary metal-oxide semiconductor

CPU – Central processing unit

dis – Disassembler

FP – Frame Pointer

FPE – Floating point emulation library

ISP – Interrupt stack pointer

ld – Link editor

LSB – Least significant bit

MAU – Math acceleration unit

mmmm – Mode field

MIS – MAU instruction set

MSB – Most significant bit

N – Condition flag bit negative

NMOS – N-channel metal-oxide semiconductor

PC – Program counter

PCBP – Process control block pointer

PSW – Processor status word

RA – Return address

rrrr – Register field

SP – Stack pointer

V – Condition flag bit overflow

Z – Condition flag bit zero

Index

NOTES

NOTES