IBM RT PC Advanced Interactive Executive Operating System

Assembler Language Reference

Programming Family

Personal
Computer
Software

59X7994

IBM RT PC Advanced Interactive Executive Operating System

Assembler Language Reference

Programming Family

IBM

Personal
Computer
Software

First Edition (November 1985)

Changes are made periodically to the information herein; these changes will be incorporated in new editions of this publication.

References in this publication to IBM products, programs, or services do not imply that IBM intends to make these available in all countries in which IBM operates. Any reference to an IBM program product in this publication is not intended to state or imply that only IBM's program product may be used. Any functionally equivalent program may be used instead.

International Business Machines Corporation provides this manual "as is," without warranty of any kind, either express or implied, including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this manual at any time.

Products are not stocked at the address given below. Requests for copies of this product and for technical information about the system should be made to your authorized IBM RT PC dealer.

A reader's comment form is provided at the back of this publication. If the form has been removed, address comments to IBM Corporation, Department 997, 11400 Burnet Road, Austin, Texas 78758. IBM may use or distribute whatever information you supply in any way it believes appropriate without incurring any obligation to you.

About This Book

This book discusses the RT PC implementation of the 032 Microprocessor assembler language on the AIX Operating System. The book contains 032 Microprocessor assembler language syntax and semantics, along with information on linking and running an 032 Microprocessor assembler language program. The book also shows how to link to programs written in the high-level languages supported by RT PC.

Programmers may prefer to use a high-level language for general-purpose programming instead of 032 Microprocessor assembler language because:

- The 032 Microprocessor has a reduced instruction set.
- The 032 Microprocessor was designed as an efficient target for compilers, but not for assembler language programming. For example, the assembler does not give an error message for undefined external symbols.

Note that this book does not teach assembler language programming. You should use this book for reference only.

Who Should Read This Book

This book is intended for applications and systems programmers who know another assembler language. This book contains information about using 032 Microprocessor assembler language to write portions of application programs. For example, it might be appropriate to use the 032 Microprocessor assembler language to create small library routines to take advantage of architectural functions not available in the C language.

The 032 Microprocessor assembler language also allows you to write code for the Virtual Resource Manager (if it exists) and the AIX Operating System kernel. This book includes the processor instructions to perform these tasks, but does not discuss them in detail. If you want to write 032 Microprocessor assembler language programs to perform these tasks, or if you want information about RT PC architecture, you need to read books in addition to this one. (See "Related Publications" on page viii and "What Is Not in This Book" on page vi.)

This book assumes that you know how to use your RT PC system. You should be able to log in, create files, edit files, and use various other operating system commands. If you need information about these topics, see "Prerequisite Publications" on page vii.

How This Book is Organized

"Chapter 1. Overview of Processing and Storage on the 032 Microprocessor" briefly explains data representation and registers on the 032 Microprocessor.

"Chapter 2. Assembler Language Concepts" discusses the syntax of statements, expressions, symbols, and constants. This chapter also explains the 032 Microprocessor assembler language's character set, along with the notational conventions used in this book.

"Chapter 3. Addressing and Program Sectioning" explains how to combine lines of assembler code. This chapter includes information on addressing and on declaring base registers. This chapter also explains the assembler's location counter and the relationship between **a.out** segments and assembler language sections.

"Chapter 4. 032 Microprocessor Instructions" describes the 032 Microprocessor instructions with both mnemonics and op codes. The instructions are listed by function, then described alphabetically by mnemonic.

"Chapter 5. Pseudo-Ops" provides an alphabetical description of the directives programmers can send to the assembler itself.

"Chapter 6. Assembling, Linking, and Running a Program" describes the AIX Operating System commands used to assemble, link, and run assembler language files. This chapter also describes the conventions you should use to link 032 Microprocessor assembler language files to files written in other languages.

"Appendix A. Mnemonic and Op Code Tables" contains two tables. One table lists instructions alphabetically by mnemonic; a second table lists instructions numerically by op code.

"Appendix B. ASCII Character Codes" lists the numeric representations of valid characters on the RT PC.

The "Glossary" defines terms that are specific to 032 Microprocessor assembler language and to RT PC.

A Reader's Comment Form and Book Evaluation Form are provided at the back of this book. Use the Reader's Comment Form at any time to give IBM information that may improve the book. After you become familiar with the book, use the Book Evaluation Form to give IBM specific feedback about the book.

What Is Not in This Book

As mentioned previously, this book does not teach readers how to program or to operate their RT PC system. Furthermore, this book contains little or no information about:

- Any commands, system calls, subroutines, or programming aids that are part of the AIX Operating System, except for the **as** (assembler) command; limited information is given about the **cc** (C compiler) and **ld** (link editor) commands.
- Error messages generated by **as**. These messages are shown in *Messages Reference.*
- Specific aspects of creating or running any code that exists in the Virtual Resource Manager (VRM).
- Any hardware features (except some registers).
- Details about 032 Microprocessor privileged instructions, including information about I/O and supervisor calls.
- Running 032 Microprocessor assembler language programs directly on the VRM, or creating programs that run directly on the VRM.
- Running 032 Microprocessor assembler language programs on any operating system other than AIX Operating System, or creating programs that run on any such operating system.

For information about these topics, see "Related Publications" on page viii.

Prerequisite Publications

You should be familiar with the following books before you try to use this book:

- *IBM RT PC Using and Managing the AIX Operating System* describes using AIX Operating System commands, working with the file system, and developing shell procedures. This book also provides instructions for performing such system management tasks as adding and deleting user IDs, creating and mounting file systems, backing up the system, and repairing file system damage.
- *IBM RT PC Guide to Operations* describes the Model 10, Model 20, and Model 25 system units, the displays, keyboard, and other devices that can be attached. This guide also includes procedures for operating the hardware and moving the Model 10, Model 20, and Model 25 system units.
- *IBM RT PC AIX Operating System Commands Reference* lists and describes the AIX Operating System commands.

Related Publications

This book refers to the following books:

- *IBM RT PC AIX Operating System Programming Tools and Interfaces* describes the programming environment of the AIX Operating System and includes information about using the operating system tools to develop, compile, and debug programs. In addition, this book describes the operating system services and how to take advantage of them in a program. This book also includes a diskette that includes programming examples, written in C language, to illustrate using system calls and subroutines in short, working programs. (Available optionally)
- *IBM RT PC Virtual Resource Manager Technical Reference* describes the Virtual Resource Manager (VRM) routines, how to use the VRM debugger, how to develop and install code into the VRM, and defines the interface to the VRM-supplied device drivers. This book also describes the Virtual Machine Interface (VMI) between the Virtual Resource Manager and the AIX Operating System and provides information about process control, memory management, the I/O subsystem, the minidisk manager, and device drivers. (Available optionally)

  The VRM section defines VRM routines, explains how to use the VRM debugger, explains how to develop and install code into the VRM, and defines the Virtual Machine Interface (VMI) to the VRM-supplied device drivers. The VMI section describes the interface between the RT PC and the VRM, and discusses process control, memory management, the I/O system, the minidisk manager, and device drivers.
- *IBM RT PC Hardware Technical Reference* is a two-volume set. Volume I describes how the system unit operates, including I/O interfaces, serial ports, memory interfaces, and CPU interface instructions. Volume II describes adapter interfaces for optional devices and communications and includes information about IBM Personal Computer family options and the adapters supported by Model 10, Model 20, and Model 25. (Available optionally)
- *IBM RT PC AIX Operating System Technical Reference* describes the system calls and subroutines that a C programmer uses to write programs. This book also provides information about the AIX file system, special files, miscellaneous files, and writing device drivers. (Available optionally)

  Both volumes may be relevant to programmers who are writing device drivers.
- *IBM RT PC C Language Guide and Reference* provides guide information for writing, compiling, and running C language programs and includes reference information about C language data structures, operators, expressions, and statements. (Available optionally)
- *IBM RT PC Problem Determination Guide* provides instructions for running diagnostic routines to locate and identify hardware problems. Also includes problem determination for software. Two high-capacity (1.2MB) diskettes containing the IBM RT PC diagnostic routines are included.

- *IBM RT PC Messages Reference* lists messages displayed by the IBM RT PC and explains how to respond to the messages.
- *IBM RT PC Bibliography and Master Index* provides brief descriptive overviews of the books and tutorial program that support the IBM RT PC hardware and the AIX Operating System. In addition, this book contains an index to the RT PC and AIX Operating System library.

Ordering Additional Copies of This Book

To order additional copies of this publication, use either of the following sources:

- To order from your IBM representative, use Order Number SV21-8011.
- To order from your IBM dealer, use Part Number 55X-8930.

A binder is included with the order.

Contents

Chapter 1. Overview of Processing and Storage on the 032 Microprocessor

CONTENTS

About This Chapter

The RT PC 032 Microprocessor assembler language is influenced by the characteristics of its machine's storage and processor. The capabilities of the processor and the nature of the available storage determine what the assembler language can do.

This chapter gives an overview of the 032 Microprocessor and tells you how data is stored on the RT PC in main storage and in registers. This information may help you understand how to use 032 Microprocessor instructions and pseudo-ops.

The 032 Microprocessor and its Protection States

The 032 Microprocessor is a 32-bit ***pipelined*** processor. The 032 Microprocessor has a reduced instruction set, with few memory access instructions. Except for the memory access instructions, most instructions execute in a single processor cycle.

The 032 Microprocessor's assembler language is designed to support systems and application programming in high-level languages. Therefore, the 032 Microprocessor assembler language is intended primarily as an efficient target for compilers.

032 Microprocessor, the physical or "real" machine, has two protection states: privileged and unprivileged.

In privileged state, 032 Microprocessor accepts all instructions, including ***privileged instructions***. Privileged instructions generally manipulate virtual machines (if they exist) or the memory manager, and are not discussed at length in this book. 032 Microprocessor assembler language programmers can access 032 Microprocessor privileged state with the Supervisor Call (**svc**) instruction.

The 032 Microprocessor's privileged state supports virtual machines. On RT PC, the Virtual Resource Manager (VRM) is the only entity that runs in privileged state. The VRM is a collection of processes, interrupt handlers, and runtime routines that support the Virtual Machine Interface by issuing privileged instructions to the hardware. The VRM redefines the processor's physical protection states for virtual machines and creates software emulations of the processor's physical registers. The Virtual Machine Interface (VMI) is the link between the operating system and the VRM programs that regulate system resources. See *Virtual Resource Manager Technical Reference* for information about how virtual machines work on RT PC.

In unprivileged state, 032 Microprocessor accepts only ***unprivileged instructions*** such as adds, shifts, and loads. Both the operating system and ordinary application programs run in unprivileged state. Virtual machines also run in 032 Microprocessor unprivileged state.

This book includes both privileged and unprivileged instructions, but emphasizes the unprivileged instructions. For more information on the context in which privileged instructions occur, see *Virtual Resource Manager Technical Reference.*

Note: Floating point support is available with optional hardware, or with floating point software emulation in the AIX Operating System. See *Hardware Technical Reference* for information on the floating point registers.

Data Representation in Main Storage

On the RT PC, main storage is organized as a series of eight-bit bytes with a maximum address of 2^{32} - 1. Bytes in main storage are consecutively numbered, left to right, starting with zero. Four bytes make a word. The leftmost bit of a word is the high-order bit, or the most significant bit. The highest-order bit of a word is stored at a lower address in main storage than the lowest-order bit of the same word.

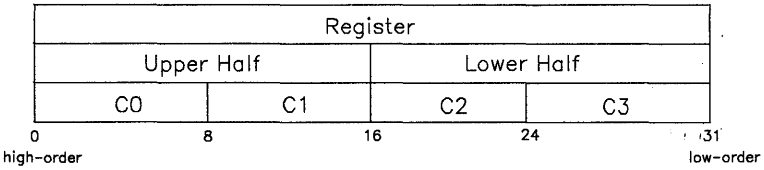


Figure 1-1. Data Units in Main Storage. The highest-order byte, C0, is the first character.

The address of a word or halfword in main storage is the address of its leftmost byte. A word must lie on a 4-byte boundary; that is, the address of a word has zeroes in the two low-order bits. A halfword must lie on a 2-byte boundary; that is, the address of a halfword has a zero in the low-order bit.

All instructions must be located on halfword boundaries.

All storage accesses are for a byte or multiple bytes. Storage accesses for words ignore the low-order pair of bits of the effective address. Storage accesses for halfwords ignore the low-order bit of the effective address.

The assembler may give you an error message if you try to access data from or store data to an invalid memory location. Invalid memory accesses will also trigger a program check (see "Program Check Status (PCS)" on page 1-11). You especially need to watch for invalid memory locations if you use the absolute branch instructions, an **.org** pseudo-op with an absolute operand, or the **.direct** pseudo-op. For information on invalid memory locations, see Chapter 3.

Wraparound is allowed and occurs on a 32-bit basis. That is, main storage addressing wraps around from the architectural maximum byte address to address 0. For example, if an instruction added 5 to the maximum byte address, the result would be address 4.

The RT PC 032 Microprocessor assembler language supports effective addresses, that is, a base address plus a displacement. Depending on the instruction, the displacement is specified as immediate data or as the contents of a register. All effective addresses are computed as 32-bit quantities. See Chapter 3 for more information on addressing.

Data Representation in Registers

The 032 Microprocessor has sixteen 32-bit general purpose registers and sixteen system control registers.

General Purpose Registers

Instructions are provided to load or store a single character, word, or halfword into a ***general purpose register*** (GPR).

Each GPR consists of an upper and lower half of sixteen bits each. Each GPR may be separated into four eight-bit characters: C0, C1, C2, and C3. The organization of General Purpose Registers is shown in Figure 1-1 on page 1-5.

For computational purposes, the content of a GPR is treated as a signed algebraic quantity, an unsigned positive quantity, or an unstructured logical quantity, depending on the instruction that does the computation. In a GPR, an algebraic quantity is represented by 32 bits in two's complement form.

Since each register is 32 bits long, the largest positive signed number that can fit in a register is decimal 2^{31} - 1. The largest negative number that can fit in a register is -2^{31}. Numbers larger than these are wrapped around. (See "Arithmetic Constants" on page 2-16.)

GPRs are manipulated by most of the instructions in the 032 Microprocessor instruction set. For information about what these instructions are and how they affect GPRs, see Chapter 4.

To avoid the destruction of operands, the following instructions cause the result of the operation to be placed in the pair of one of the GPR operands:

slp	**slpi**
srp	**srpi**

The pair of a given GPR has the name, in binary, of the given GPR with the low-order bit inverted. In this way, a pair consists of an even-numbered and an odd-numbered register. For example, GPR 5 (binary 0101) and GPR 4 (binary 0100) are pairs, and GPR 14 (binary 1110) and GPR 15 (binary 1111) are pairs. (See Figure 1-2.)

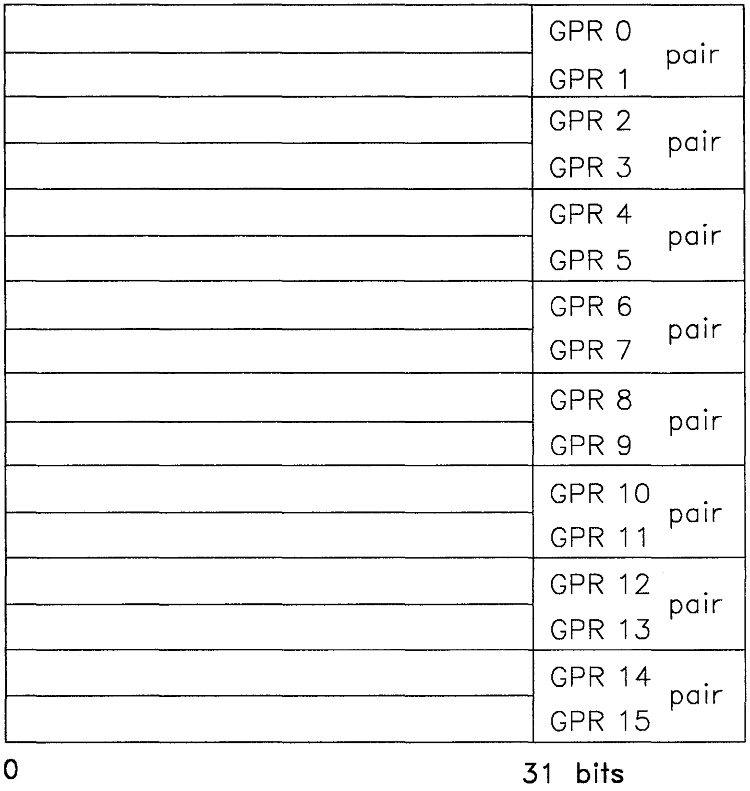


Figure 1-2. General Purpose Register Pairs

System Control Registers

The 032 Microprocessor has sixteen 32-bit ***system control registers*** (SCRs). An entire SCR or fields within an SCR are generally set aside to be used by programs that control the system. Figure 1-3 on page 1-8 shows the different SCRs.

Some SCRs and bits inside SCRs are reserved and are not assigned to any system facility. Any attempt to set the reserved bits of an SCR will be ignored. When the reserved bits of an SCR are fetched, the resulting values are unpredictable.

Reserved				SCR 0
Reserved				SCR 1
Reserved				SCR 2
Reserved				SCR 3
Reserved				SCR 4
Reserved				SCR 5
Counter Source				SCR 6
Counter				SCR 7
Reserved			TS	SCR 8
Reserved				SCR 9
Multipler Quotient				SCR 10
		MSC	PCS	SCR 11
Reserved		IRB		SCR 12
Instruction Address Register				SCR 13
Reserved		ICS		SCR 14
Reserved		CS		SCR 15
0	8	16	24	31

TS = Timer Status
MCS = Machine Check Status
PCS = Program Check Status

IRB = Interrupt Request Buffer
ICS = Interrupt Control Status
CS = Condition Status

Figure 1-3. System Control Registers

The SCRs can be manipulated with certain privileged instructions. (See "System Control Register Manipulation Instructions" on page 4-20.) Two of the SCRs can also be manipulated in unprivileged state: Multiplier Quotient (SCR 10) and Condition Status (SCR 15). The Instruction Address Register (SCR 13) cannot be directly manipulated in unprivileged state, but is affected by assembler instructions.

The following descriptions apply to SCRs in hardware. Virtual machines do not have hardware SCRs, but they have analogous virtual machine control registers. See *Virtual Resource Manager Technical Reference* for information about how virtual machines define virtual machine control registers.

Note: The processor dynamically changes SCRs, often asynchronously to instruction sequencing. Therefore, if you write to an SCR, then read from the same SCR, you may not read back the same data that you wrote in.

Counter Source, Counter, and Timer Status Registers

SCRs 6 and 7 and the TS bits of SCR 8 help control the system timer. For more information on SCRs 6 and 7, see *Hardware Technical Reference.* For more information on SCR 8, see *Virtual Resource Manager Technical Reference.*

Multiplier Quotient (MQ) Register

The MQ register holds the product of the Multiply Step (**m**) instruction or the dividend of the Divide Step (**d**) instruction. You can also manipulate the contents of the MQ register with the Move to SCR (**mts**) and Move From SCR (**mfs**) instructions. For more information about these instructions, see Chapter 4.

Note: The MQ is not preserved across subroutine calls. See "Subroutine Linkage and System Calls" on page 6-10 for more information.

Machine Check Status (MCS)

The Machine Check Status occupies bits 16 through 23 of SCR 11. When a machine check error is detected by hardware, appropriate bits of the MCS are set to one. You cannot clear a Machine Check Level by clearing the bits in the MCS, nor can you cause a Machine Check by writing to the MCS. The MCS is only cleared when a Load Program Status (**lps**) instruction is executed to return from a Machine Check Level. The MCS includes the following bits:

- Processor channel check
- Parity check
- Instruction timeout
- Data timeout
- Processor channel timeout
- I/O trap.

For more information, see *Hardware Technical Reference.*

Program Check Status (PCS)

The Program Check Status occupies bits 24 through 31 of SCR 11. The PCS provides a means for reporting the following programming errors:

- Program check with known origin
- Program check with unknown origin
- Program trap (bit 26)
- Privileged instruction exception
- Illegal op code
- Instruction address exception
- Data address exception.

Upon detection of a program check error, all bits of the PCS are cleared to zeroes. The appropriate bits of the PCS are then set to ones. You cannot clear a Program Check Level by clearing the bits in the PCS, nor can you cause a Program Check by writing to the PCS.

For more information, see *Hardware Technical Reference.*

Interrupt Request Buffer (IRB)

Bits 16 through 31 of SCR 12 are the Interrupt Request Buffer. The IRB allows different levels of interrupt requests to be generated under program control. The interrupt request remains active until the bit is cleared by software.

For more information, see *Hardware Technical Reference.*

Instruction Address Register (IAR)

The Instruction Address register, sometimes called the instruction pointer or program counter, contains the address of the next instruction to be executed. During instruction execution, the IAR is incremented, in bytes, by the length of the current instruction. If this instruction is a successful branch, the IAR is set to the address of the branch target instruction.

In privileged state, programs can use the Move to System Control Register (**mts**) instruction to load the IAR with a specified value. Application programs, however, can only change the contents of the Instruction Address register implicitly, by executing instructions.

Do not confuse the IAR with the assembler's location counter. See "The Location Counter" on page 3-11.

Interrupt Control Status (ICS)

Bits 16 through 31 of SCR 14 are the Interrupt Control Status. Bits 0 through 15 of SCR 14 are reserved. In hardware, the ICS contains the following defined bits:

- Parity error retry interrupt enable
- Storage protect
- Problem state
- Translate mode
- Interrupt mask
- Check stop mask
- Register set number
- Processor priority.

For more information on the ICS, see *Hardware Technical Reference.*

Condition Status (CS) Register

Bits 16 through 31 of SCR 15 are the Condition Status register. However, only bits 24 through 31 are defined. Bits 0 through 23 in SCR 15 are reserved.

You can manipulate the entire contents of the CS register with the Move to System Control Register (**mts**) and the Move from System Control Register (**mfs**) instructions.

You can manipulate any single bit in the CS register with the Set Bit in the System Control Register (**setcb**) and Clear Bit in the System Control Register (**clrcb**) instructions.

The following instructions allow you to test any single CS bit, then branch or not branch, depending on the value of that bit:

- Branch on Condition Bit (**bb**)
- Branch on Condition Bit Using Register (**bbr**)
- Branch on Condition Bit with Execute (**bbx**)
- Branch on Condition Bit Using Register with Execute (**bbrx**)
- Branch on Not Condition Bit (**bnb**)
- Branch on Not Condition Bit Using Register (**bnbr**)
- Branch on Not Condition Bit with Execute (**bnbx**)
- Branch on Not Condition Bit Using Register with Execute (**bnbrx**).

Other instructions can test or set only one pre-defined bit in the CS register.

Bits 24 through 31 of the Condition Status register are defined as follows:

Bit 24	Permanent Zero (PZ)
Bit 25	Less Than (LT)
Bit 26	Equal (EQ)
Bit 27	Greater Than (GT)
Bit 28	Carry Zero (C0)
Bit 29	-- Reserved --
Bit 30	Overflow (OV)
Bit 31	Test Bit (TB).

The Permanent Zero (PZ) bit is always set at zero; it cannot be set to one. The PZ bit is tested by the Branch (**b**) instruction.

The Less Than (LT) bit is set during the Compare (**c**) instruction to indicate the relative algebraic magnitudes of the numbers being compared. The LT bit is also set to one during logical, shift, and certain arithmetic instructions if the result of the instruction is negative or if the high-order bit of the result is one. Otherwise, LT is set to zero. The LT bit is tested by the Branch on Less Than (**blt**) and Branch on Greater Than or Equal (**bge**) instructions.

The Equal (EQ) bit is set during the Compare (**c**) instruction to indicate if comparands are equal. The EQ bit is also set to one during logical, shift, and certain arithmetic instructions if all bits of the result are zeros; otherwise it is set to zero. The EQ bit is tested by the Branch on Equal (**beq**) and Branch on Not Equal (**bne**) instructions.

The Greater Than (GT) bit is set during the Compare (**c**) instruction to indicate the true relative algebraic magnitudes of the comparands. The GT bit is also set to one during logical, shift, and certain arithmetic instructions if the sign bit of the result is zero and the result is non-zero. The GT bit is tested by the Branch on Greater Than (**bgt**) and Branch on Less Than or Equal (**ble**) instructions.

The Carry Zero (C0) bit is set to one during certain arithmetic instructions if the operation generates a carry out of a bit position zero; otherwise it is set to zero. The C0 bit is tested by the Branch and Carry Bit Set (**bcs**) and Branch and Carry Bit Clear (**bcc**) instructions.

The Overflow (OV) bit is set to one during certain arithmetic instructions if the signed result of the operation cannot be represented in 32 bits; otherwise it is set to zero. The OV bit is tested by the Branch on Overflow Set (**bvs**) and Branch on Overflow Clear (**bvc**) instructions.

The Test (TB) bit is set by the Move to Test Bit instructions (**mttb** and **mttbi**), where a specified bit of a register is moved to the test bit. It is also affected by instructions which load or directly alter data in the Condition Status register. The TB bit is tested by the Branch on Test Bit Set (**bts**) and Branch on Test Bit Clear (**btc**) instructions.

Chapter 2. Assembler Language Concepts

CONTENTS

About This Chapter

This chapter explains the syntax and semantics of 032 Microprocessor assembler language. It tells you what you need to know to write a single line of 032 Microprocessor assembler language code.

The following topics are discussed:

- Notational conventions used to describe syntax
- Character set
- Reserved words
- Line format
- Components of a statement
- Kinds of statements
- Symbols
- Constants
- Operators and expressions.

For information on combining lines of 032 Microprocessor assembler language code (addressing, segmenting, the location counter), see Chapter 3.

Notational Conventions

Throughout this chapter, certain notational conventions describe aspects of the 032 Microprocessor assembler language.

- All spaces are required unless otherwise noted.
- In commands, capitalization is significant. All uppercase and lowercase letters should be entered exactly as shown.
- Optional parts of statements are shown in brackets. However, you do not type the brackets themselves. For example,

  `mnemonic operand1[,operand2]`

  means that you can type

  `mnemonic operand1` or

  `mnemonic operand1,operand2` .

- Brackets can be nested. For example,

  `mnemonic [operand1[,operand2]]`

  means that you can type

  `mnemonic` or

  `mnemonic operand1` or

  `mnemonic operand1,operand2` .

Character Set

All letters and numbers are allowed. However, the assembler discriminates between upper-case and lower-case letters. For example, the assembler sees the symbol `Name` as being different from the symbol `name`.

Some blank spaces are required; others are optional. (See "Statements" on page 2-9.) The assembler allows you to substitute tabs for spaces.

The following characters have a special meaning in 032 Microprocessor assembler language:

, (comma)

Operand separator. Commas are allowed only between operands. (See "Statements" on page 2-9.)

Example:

```
lr 2,15
```

# (pound sign)

Comments. Anything from # to the end of a line is a comment and is ignored by the assembler. A # can be the first character in a line, or it may be preceded by any number of characters, blank spaces, or both. (See "Comments" on page 2-11.)

Example:

```
lr 2,15    # loads register 2 with contents
           # of register 15
```

: (colon)

Defines a label equal to the value contained in the current location counter. Always appears immediately after the last character of the label name. (See "Labels" on page 2-10.)

Example:

```
here: lr 2,15  # makes here equal to the address
               # where the lr instruction is assembled
```

; (semicolon)

Instruction separator. A semicolon separates two instructions that appear on the same line. Spaces around the semicolon are optional.

A single instruction on one line does not have to end with a semicolon.

Example:

```
lr 2,15                 # these two lines have
lr 15,3                 # the same effect as...

lr 2,15 ; lr 15,3       # ...this line
```

$ (dollar sign)

Refers to the current value of the assembler's current location counter. (See "The Location Counter" on page 3-11.)

Example:

```
dino:   .long 1,2,3
size:   .long $ - dino
```

Reserved Words

032 Microprocessor assembler language does not have any reserved words. The mnemonics for instructions and pseudo-ops are not reserved; they may be used in the same way as any other symbols.

There may be restrictions on the names of symbols that are passed to programs written in other languages. For information on these restrictions, see "Subroutine Linkage and . System Calls" on page 6-10.

Line Format

On the RT PC, 032 Microprocessor assembler language is written in free format. There are no requirements for certain things to be in any particular column position.

The assembler language puts no limit on the number of characters that can appear on a single input line. If you write a longer code line than can display as one line on your terminal, line wrapping will depend on the editor you are using.

Blank lines are allowed; the assembler ignores them.

Statements

032 Microprocessor assembler language has three kinds of statements: instruction statements, pseudo-operation statements, and null statements.

Instruction Statements and Pseudo-Operation Statements

An instruction or pseudo-op statement has the following syntax:

[label:]mnemonic[operand1[,operand2]] [#comment]

The assembler recognizes the end of a statement when one of the following occurs:

- An ASCII newline character
- A comment character (#)
- A semicolon (;) .

Separator Character

The separator characters are spaces, tabs, and commas. Commas separate operands. Spaces or tabs separate the other parts of a statement. A tab can be used wherever a space is shown in this book.

The spaces shown are required spaces. You can optionally put one or more spaces after a comma, before a pound sign (#), and after a #.

Labels

The label entry is optional. A line may have zero, one, or more labels. A line may have a label but no other contents.

To define a label, follow a symbol with a colon. The assembler gives the label the value contained in the assembler's current location counter. This value represents a relocatable address.

See "Symbols" on page 2-13 for information on defining symbols.

Example 1

```
here: lr 2,15
       # the label here receives a value
       # of the address of the lr instruction.
       # You can now use here in subsequent statements
       # to refer to this address.
```

If the label is in a statement with an instruction that causes data alignment, the label receives its value before the alignment occurs.

Example 2

```
    # assume that the location counter now
    # contains the value of 98

place: .long expr

    # When the assembler sees this statement, it
    # sets place to address 98.  But the
    # .long is a pseudo-op that
    # aligns expr on a fullword.  Thus,
    # the assembler puts expr at the next
    # available fullword boundary, which is
    # address 100.  In this case place is
    # not actually the address at which expr
    # is stored; referring to place will not
    # put you at the location of expr.
```

Mnemonics

The mnemonic field identifies whether a statement is an instruction statement or a pseudo-op statement. Each mnemonic requires a certain number of operands in a certain format.

For an instruction statement, the mnemonic field contains an abbreviation like **ais** or **beq**. This mnemonic describes an operation where the 032 Microprocessor processes a single machine instruction, which is associated with a numerical op code. Instructions vary in length, so the op code tells the assembler how long the instruction and its operands are going to be. When the assembler encounters an op code, the assembler increments the location counter by the required number of bytes.

For a pseudo-op statement, the mnemonic represents an instruction to the assembler program itself. There is no associated op code, and the mnemonic does not describe an operation to the processor. Some pseudo-ops increment the location counter; others do not.

Operands

The existence and meaning of the operands depends on the mnemonic used. Some mnemonics do not require any operands. Other mnemonics require one or more operands.

The assembler interprets each operand in context with the operand's mnemonic. Many operands are expressions that refer to registers or symbols. For instruction statements, operands can be immediate data that is to be directly assembled into the instruction.

Comments

Comments are optional and are ignored by the assembler. Every line of a comment must be preceded by a pound sign (#); there is no other way to designate comments.

Null Statements

A null statement does not have a mnemonic or any operands. It can contain a label, a comment, or both. Processing a null statement does not change the value of the location counter.

Null statements are useful mainly to make assembler source code easier for people to read.

A null statement has the following syntax:

```
[label:]      [#comment]
```

The spaces between the label and the comment are optional.

If the null statement has a label, the label receives the value of the next statement, even though that statement is on a different line. For example,

```
here:
call6 3,'X
```

has the same effect as

```
here: call6 3,'X
```

Note: Certain pseudo-ops may prevent a null statement's label from receiving the value of the next statement. See Example 2 on page 2-10.

Symbols

A symbol is a single character or combination of characters used as a label or operand. Symbols may consist of numeric digits, underscores, periods, upper or lower case letters, or any combination of these. The symbol cannot contain any blanks or special characters, and cannot begin with a digit. Upper and lower case are distinct.

From the assembler and loader's perspective, the length of a symbol name is limited only by the amount of storage you have. However, only the first 32 characters are significant. Also note that other routines linked to assembler language files may have their own constraints on symbol length.

You can use a symbol to represent storage locations or arbitrary values. The value of a symbol is always a 32-bit quantity.

The following are examples of valid symbol names:

```
READER
A2345
result.a
resultA
balance_old
_label9
.myspot
```

The following are examples of invalid symbol names:

`7_sum`	(begins with a digit)
`#ofcredits`	(contains #, a special character)
`aa*1`	(contains *, a special character)
`IN AREA`	(contains a blank)

You can define a symbol by using it in one of two ways:

- As a label for an instruction or pseudo-op
- As the **name** operand of a **.set**, **.comm**, or **.lcomm** pseudo-op.

Defining a Symbol with a Label

You can define a symbol by using it as a label.

Example:

```
          bali      1,cont
          .using    $,1
dataval:  .short    10
   .
   .
cont:     lh        2,dataval
          a         4,2
```

The assembler gives the symbol the value of the location counter at the instruction or pseudo-op's leftmost byte. In the example above, the object code for the **lh** instruction contains the location counter value for **dataval**.

At runtime, this value represents an address, and the contents of that address are used as an operand. In the example above, the **lh** instruction uses the 16 bits of data stored at **dataval**'s address.

Note that the value referred to by the symbol actually occupies a memory location. A symbol defined by a label is a relocatable value.

The symbol itself does not exist at runtime. However, you can change the value at the address represented by a symbol at runtime, if some code changes the contents represented by **dataval**.

Defining a Symbol with a Pseudo-op

You can also define a symbol by using it as the **name** operand of a **.set** pseudo-op. This pseudo-op has the format **.set name, exp**. For example:

```
.set number,10
     .
     .
ais 4,number
```

The assembler evaluates the **exp** operand, then assigns the value and type of **exp** to the symbol **name**. When the assembler encounters that symbol in an instruction, the assembler puts the symbol's value into the instruction's object code. In the example above, the object code for the **ais** instruction contains the value assigned to **number**, that is, 10.

Note that the value of the symbol is assembled directly into the instruction, and does not occupy any storage space. A symbol defined with a **.set** can have an absolute or relocatable type, depending on the type of the **exp** operand. (See "Types of Expressions" on page 2-20.) Also, you cannot change the value of the symbol at runtime; you must reassemble the file in order to give the symbol a new value.

You can also define a symbol by using it as the **name** operand of a **.lcomm** or **.comm** pseudo-op. In this case, the value assigned to the symbol does occupy storage space; see Chapter 5.

Using A Symbol Before Defining It

It is possible to use a symbol before you define it. Using a symbol, and then defining it later in the same file, is called forward referencing. In other words, the following is acceptable:

```
    l 5,ten
       .
       .
ten: .long 10
```

If the symbol is not defined in the file in which it occurs, it is called an external symbol. When the assembler finds external symbols, it does not give you an error message; it assumes that you will link in another file that defines the symbol.

For branch instructions only, the symbol and its definition must be, after linking, in the same runtime segment as all references to that symbol. (See Chapter 3 for information about runtime segments and their relation to assembler language sections.)

The only exception is for symbols that have been the subject of a **.direct** pseudo-op, which allows you to make a direct reference to an external symbol. If you use **.direct**, the symbol you refer to must occupy an address in the lowest 32K of memory at link time.

Constants

The 032 Microprocessor assembler language has three kinds of constants:

- Arithmetic constants
- Character constants
- Symbolic constants (symbols being used as constants).

When the assembler encounters an arithmetic or character constant that is being used as an instruction's operand, the value of that constant is assembled into the instruction. (This is why arithmetic and character constants are sometimes called self-defining terms.) When the assembler encounters a symbol being used as a constant, the value of the symbol is assembled into the instruction. (Symbolic constants are sometimes called data constants or ordinary symbols.)

The assembler eventually translates all constants, no matter how they are specified, into 32-bit integer constants.

Arithmetic Constants

There are three kinds of arithmetic constants: decimal, octal, and hexadecimal.

Because the 032 Microprocessor is a 32-bit processor, the largest signed positive number that any single register can hold is the decimal value 2^{31}-1. The largest negative value allowed in a register is -2^{31}. If you specify an expression that evaluates to a constant with a value larger then these, the value wraps around at runtime. For example,

```
0x7FFFFFF8 + 0x14
```

yields a value of 0x8000000C (a large negative number).

The largest unsigned number that any single register can hold is 2^{32}-1. For unsigned numbers, wraparound works like this:

```
0xFFFFFFF8 + 0x14
```

yields a value of 0x0000000C.

Decimal Constants

Base 10 is the default base for arithmetic constants. If you want to specify a decimal number, just type the number in the appropriate place.

```
  ai 5,10  # adds decimal value 10 to contents of GPR 5
```

Do not prefix decimal numbers with a zero. A leading zero denotes an octal number.

Octal Constants

To specify an octal number, prefix the number with the numeral **0**.

```
ai 5,0377  # adds octal value 377 to contents of GPR 5
```

Hex Constants

To specify a hexadecimal number, prefix the number with **0X** or **0x**. You can use either uppercase or lowercase for the hex numerals A through F.

```
ais 5,0xF  # adds hex value F to GPR 5
ais 3,0X5  # adds hex value 5 to GPR 3
```

Character Constants

To specify an ASCII character constant, prefix the constant with a ' (single quote mark). Character constants can appear anywhere an arithmetic constant is allowed, but you can only specify one character constant at a time. For example, **'A** represents the ASCII code for the character A.

Character constants are convenient when you want to use the code for some character as a constant. For example,

```
cal16 3,'X  # Loads GPR 3 with the ASCII code for
            # the character X (that is, hex 58).
            # After the cal16 instruction
            # executes, the low-order 16 bits of
            # GPR 3 contain
            # binary 0000 0000 0101 1000.
```

Symbolic Constants

All symbols do not have to be used as constants. However, 032 Microprocessor assembler language allows you to use a symbol as a constant. Once you define a value, you can refer to that value by name, instead of using the value itself.

Using a symbol as a constant is convenient if you have a value that occurs frequently in your program. You define the symbolic constant once, by giving the value a name. If you decide to change the value, you only have to change its definition, not every reference to it in the program.

A symbolic constant can be defined by using it as a label or by using it in a **.set** statement. "Symbols" on page 2-13 discusses how to define symbols.

Expressions

An expression is a constant, a symbol, or a combination of constants, symbols, and operators. The assembler evaluates each expression into a single value, then uses that value as an operand. Expressions have a type as well as a value.

Operators and Operator Precedence

032 Microprocessor assembler language allows the following operators:

( )	control order of evaluation
+	addition or unary +
-	subtraction or unary two's complement
*	multiplication
/	division
&	logical and
¦	inclusive or
^	exclusive or
<	logical left shift
>	logical right shift
~	unary bitwise complement

All these operators evaluate left to right, except for the unary operators, which evaluate right to left.

Operator precedence is as follows:

highest priority

()

unary -   unary +   ~

* / < >

¦ ^ &

+ -

lowest priority

All the operators perform 32-bit signed integer operations.

The division operator, /, produces an integer result; the remainder has the same sign as the dividend. For example,

Operation	**Result**
8/3	2
8/-3	-2
(-8)/3	-2
(-8)/(-3)	2

The left shift (<) and right shift (>) operators take an integer bit value for the right-hand operand. For example,

```
.set mydata,1
.set newdata,mydata<2   # shifts 1 left 2 bits,
                        # assigns result to newdata
```

Types of Expressions

There are three types of expressions: absolute, relocatable, or external. The type of an expression depends on the type of its operands.

Expression types are important for two reasons. First, some pseudo-ops and instructions require expressions of a particular type. Second, only certain operators are allowed in certain types of expressions, as described below.

In the explanations below, “absolute” recursively refers to an absolute expression, and “relocatable” recursively refers to a relocatable expression. “A symbol set to ...” means a symbol that has appeared in a **.set** statement (**.set name, ...**).

Absolute Expressions

The value of an absolute expression is independent of any possible code relocation. The value of an absolute expression stays the same, no matter where the runtime segment containing the expression is loaded.

Absolute expressions must be one of the following:

- An integer or character constant
- A symbol set to an absolute
- absolute <operator> absolute , where <operator> is any arithmetic binary operator
- - absolute
- ~absolute
- relocatable - relocatable, where the two “relocatables” refer to the same assembler section.

The definitions of “absolute” and “relocatable” above are recursive. For example, *absolute <operator> absolute <operator> relocatable - relocatable* is a valid absolute expression.

Any expression not covered by the above rules is invalid. An example of an invalid absolute expression is *relocatable + relocatable.*

Relocatable Expressions

The value of a relocatable expression depends on the location of the runtime segment containing the relocatable expression. If the runtime segment moves to a different storage location, the value of the relocatable expression changes accordingly.

Since the runtime segments can be relocated independently, the type of a relocatable expression includes the runtime segment. (See “Assembler Sections and Runtime Segments” on page 3-9.)

Relocatable exprpssions must be one of the following:

- A label
- A symbol set to a relocatable expression
- relocatable + absolute
- relocatable - absolute
- absolute + relocatable.

The definitions of “absolute” and “relocatable” above are recursive. For example, *absolute + (relocatable + absolute)* is a valid relocatable expression.

Any expression not covered by the above rules is invalid. Examples of invalid relocatable expressions are *relocatable*absolute*, *absolute - relocatable.*

All expressions that are based on the location counter are relocatable (for example, those used as operands for the **bnb** instruction and related instructions).

The final resolution of the value represented by a relocatable expression is performed at load time by **ld**.

External Expressions

External expressions refer to external symbols (symbols not defined in the current file).

If the external expression is used as a label, the expression is relocatable. An external expression cannot be used as the subject of a **.set**.

External expressions must be one of the following, where “external” refers to an external expression:

- A symbol declared **.comm**
- Any symbol not otherwise defined
- external + absolute
- external - absolute
- absolute + external .

The definitions of “absolute” and “external” above are recursive. For example, *absolute + (external + absolute)* is a valid external expression.

Any expression not covered by the above rules is invalid. Examples of invalid external expressions are *external + relocatable*, *absolute - external.*

Chapter 3. Addressing and Program Sectioning

CONTENTS

About This Chapter

This chapter explains some things you need to know to combine lines of 032 Microprocessor assembler language code.

The first part of the chapter discusses addressing, base registers, the **.using** and **.drop** pseudo-ops, and addresses with a special meaning. The second part of the chapter discusses assembler language sections, the assembler's location counter, and how assembler language sections map to runtime segments in the executable object file.

Addressing

Some addresses are already occupied by code that controls the RT PC system. These special addresses are discussed on page 3-7.

The 032 Microprocessor supports four basic kinds of addressing modes:

- Absolute immediate
- Absolute
- Relative immediate
- Based (short and long).

Since the first three addressing modes are used only by branch instructions, these modes are also discussed in "Branch Instructions" on page 4-9.

Absolute Immediate Addresses

An absolute immediate address is designated by immediate data. This addressing mode is absolute in the sense that it is not specified relative to the IAR.

On the 032 Microprocessor, only the **bala** and **balax** instructions have an absolute immediate addressing mode. These instructions assemble a 24-bit immediate operand which is extended on the left with eight binary zeroes to become the branch target address. The immediate operand can be an absolute, relocatable, or external expression.

Absolute Addresses

An absolute address is represented by the contents of a register. This addressing mode is absolute in the sense that it is not specified relative to the IAR.

On the 032 Microprocessor, the absolute addressing mode is used by the instructions **bbr[x], bnbr[x]**, extended branch instructions with the same op codes as these, and **balr[x]**. These instructions have a register as an operand; the contents of this register become the branch target address. The contents of the register can be determined by a relocatable, absolute, or external expression.

Relative Immediate Addresses

Relative immediate addresses are specified as immediate data within the object code, and are calculated relative to the IAR. On the 032 Microprocessor, all the instructions that use relative immediate addressing are branch instructions: **bb[x], bnb[x],** extended branch instructions with the same op codes as these, and **bali[x]**. These instructions have immediate data which is the displacement in halfwords from the current IAR. At execution, the immediate data is sign extended, logically shifted left one bit, and added to the address of the branch instruction to calculate the branch target address.

Relative immediate addresses are specified with either a 20-bit immediate field or an 8-bit immediate field, depending on the instruction. (See Chapter 4 for details.)

Based Addresses

In this book, the instructions that allow based addresses have a **D2(R2)** operand. (The 2′s indicate the second operand in an instruction.) Some instructions require **D2** to have a value that can be contained in 4 bits. Other instructions require **D2** to have a value that can be contained in 16 bits.

If an instruction does not have an operand of the form **D2(R2)**, then you cannot specify a based address for that instruction.

There are two ways to specify based addresses: explicitly, or implicitly.

Explicit Based Addresses

You write an explicit based address by specifying a base register number, **(R2)**, and a displacement, **D2** (also called an offset). The base register holds a base address. At runtime, the processor adds the displacement to the contents of the base register to obtain the effective address.

You must use an absolute expression to specify the base register itself. However, the contents of the base register can be specified by an absolute, relocatable, or external expression. If the base register holds a relocatable value, the effective address is relocatable. If the base register holds an absolute value, the effective address is absolute. If the base register holds a value specified by an external expression, the type of the effective address is absolute if the expression is eventually defined as absolute, and relocatable if the expression is eventually defined as relocatable.

The storage instructions have short and long forms, where **D2** can be either 4 or 16 bits. The assembler attempts to use a short form of a data reference wherever it can to save space. It only does so if it can determine the displacement in pass 1, that is, when the operand is an explicit based expression with a constant positive displacement.

Notes:

1. GPR 0 cannot be used as a base register. Specifying 0 tells the assembler not to use a base register at all.
2. Since **D2** occupies 16 bits at most, the maximum positive displacement is 2^{15} - 1, and the maximum negative displacement is 2^{15}. Therefore, the difference between the base address and the address of the item to which reference is made must be less than 2^{15} bytes.

Implicit Based Addresses (.using and .drop)

To specify an implicit based address as an operand for an instruction, omit the **(R2)** operand and write the **.using** pseudo-op at some point before the instruction. After assembling the appropriate **.using** and **.drop** pseudo-ops, the assembler knows the register to use as the base register. At runtime, the processor computes the effective address, just as if you had explicitly specified the base in the instruction.

Implicit based addresses can be relocatable or absolute, depending on the type of expression used to specify the contents of **R2** at runtime. Usually you specify the contents of **R2** with a relocatable expression, thus making a relocatable implicit based address. In this case, when the object module produced by the assembler is relocated, only the contents of the base register will change. The displacement remains the same, so **D2(R2)** still points to the correct address after relocation.

However, you can make an absolute implicit based address by specifying the contents of **R2** with an absolute expression. In this case, **R2** will not change when the object module is relocated.

In order to specify an implicit address, you must:

1. Write a **.using** statement to tell the assembler that one or more GPRs will now be used as base registers.
2. In this **.using** statement, tell the assembler the value each base register will contain at execution. Until it encounters a **.drop**, the assembler will use this base register value to process all instructions that require a based address.
3. Load each base register with the value you said it would have.

When you omit the **(R2)** operand, the **D2** operand remains. **D2** is a label or an expression containing a label.

Note: The **.using** and **.drop** pseudo-ops affect only based addresses.

Example of Implicit Based Addressing

```
.data
  foo: .long 2,3,4,5,6
  bar: .long 777

.text
       .align 2
       bali 10,yee
       .long foo
  yee: l 10,0(10)
       .using foo,10 # now you only need to
                     # specify displacement
       l 0,foo       # the assembler generates l 0,0(10)
       l 1,foo+4     # the assembler generates l 1,4(10)
       l 2,bar       # the assembler generates l 2,20(10)
```

Special Addresses

In virtual memory, user-made programs are always stored at hex addresses 1000 0000 or higher. However, this fact is not apparent to 032 Microprocessor assembler programs, because these programs are normally relocatable. You don't have to worry about referencing special memory locations (for example, the addresses occupied by the operating system).

The exceptions are for:

- Targets of an absolute branch-and-link instruction (**bala** or **balax**)
- Instructions covered by a **.direct** pseudo-op
- The **.org** pseudo-op when it has an absolute expression for an operand
- The load and store instructions (especially the store instructions **st**, **stc**, **sth**, and **stm**).

In these cases, you can specify addresses representing special memory locations that normally hold important pieces of code (i.e. the kernel). These addresses are not strictly reserved—you can reference them by using the appropriate instructions—but they are special in the sense that ordinary applications programs avoid them.

If you use **.direct**, an absolute **.org**, or an absolute immediate branch instruction (probably for AIX Operating System kernel programming), you need to know about special addresses. Figure 3-1 shows how virtual memory is laid out, and implies the addresses user programs must avoid.

Segment Register	**Hex Addresses**	**Purpose**
0	0000 0000 through 0FFF FFFF	holds AIX Operating System kernel
1	1000 0000 through 1FFF FFFF	holds program text
2	2000 0000 through 2FFF FFFF	holds program data
3	3000 0000 through 3FFF FFFF	holds the stack
4 through 13	4000 0000 through DFFF FFFF	holds shared data segments
14	E000 0000 through EFFF FFFF	reserved by Virtual Resource Manager for direct memory access I/O
15	F000 0000 through FFFF FFFF	used to map I/O bus

Figure 3-1. Segment registers and their contents

Note: Programs must not access locations in the stack segment that are below the stack floor. See “The Stack Floor” on page 6-14.

Assembler Sections and Runtime Segments

An 032 Microprocessor assembler language file may have up to three sections, which are specified with pseudo-ops:

- Text (for instructions)
- Data (for initialized data)
- Bss (for uninitialized data).

When you assemble a program with **as** or link edit a program with **ld**, the output goes into an object file called, by default, **a.out**. This executable object module has a fixed format, with segments that correspond to the text and data sections declared within an assembly language file. Assembler language sections are thus represented at runtime by **a.out** segments.

When the linker links two or more assembly language files, the linker puts the assembler sections together in the order required by the **a.out** format, even if parts of a section came from different files.

Sectioning may also be convenient for assembly language programmers. Without sectioning, you must write all the instructions in one chunk, and then write all the data. With sectioning, you can write instructions, data, then more instructions.

The text and data assembler language sections each have four location counters. These allow you to divide any section into up to four parts. You can specify the location counter with pseudo-ops. (See "The Location Counter" on page 3-11.)

Notes:

1. The branch instructions only allow branching within a segment, not branching between segments. (See "Branch Instructions" on page 4-9.)
2. For more information on **a.out**, including the use of segment registers to locate information in memory, see *AIX Operating System Technical Reference.*

The Text Section and Text Segment

The assembler text section holds the instruction and pseudo-op statements that control the program's execution. By default, the assembler assumes **.text 0**, but you can explicitly declare the text section in the assembler source code by using the **.text** pseudo-op.

The linked text sections of the assembly language file become the text segment of the executable **a.out** file.

Notes:

1. If the **.direct** pseudo-op occurs in the source program, the assembler assumes that the program's text segment will be linked in the lowest 32K of address space. This feature may help kernel programmers by allowing 16-bit direct addresses to be used for text section references not covered by **.using** statements. (See "Special Addresses" on page 3-7.)
2. If you want your program to include traceback information for debuggers such as **sdb**, the text section must end with certain lines of assembler language code. See "Traceback" on page 6-21 for details.

The Data Section and Data Segment

The data section of an assembler language program holds the data that will become the object module's initialized data. You declare the data section with the **.data** pseudo-op. The **.data** is typically followed by data alignment pseudo-ops such as **.byte** and **.long**.

The linked data sections of the assembly language file become the data segment of the **a.out** file.

The Bss Section in the Data Segment

At runtime, the bss (block startup by symbol) section is space reserved for uninitialized external values. The bss section lies at the end of the data segment. At assembly, the bss section has space reserved for it, but it contains no values. Therefore, a bss section *per se* does not exist in the assembler language source file.

However, you should use the **.lcomm** pseudo-op to reserve space in the bss section. Unlike **.text** and **.data**, **.lcomm** is not followed by any instructions or data. The **.lcomm** pseudo-op simply tells the assembler to reserve an area.

Information that will be in the bss section is not part of the **a.out** file. However, **.lcomm** does pass information about the bss size via the **a.out** file's header.

The **.comm** pseudo-op also affects the bss section. With **.comm**, you define a common block with a symbol name. The linker then defines common blocks in the bss section of a linked program, unless you link in a module that defines the symbol. If you declare a common storage area with **.comm** without declaring it as a global label in the same or another file, the loader allocates memory from the bss section, and uses the largest size declared.

The Location Counter

Each section of an assembler language program has four location counters that assign storage addresses to your program's statements. As the instructions of a source module are being assembled, the location counter keeps track of locations in storage. You can use a dollar sign ($) as an operand to refer to the current value of the location counter.

As each statement is read, the assembler increments the location counter in the following fashion:

1. After an instruction has been assembled, the location counter indicates the next available instruction. The next available instruction becomes the current instruction.
2. If the statement containing the current instruction has a label, the assembler gives this label the current value of the location counter.
3. Before assembling the current instruction, the assembler checks the boundary alignment for that instruction. If the instruction needs to be aligned, the assembler increments the location counter to indicate the proper boundary.
4. While the instruction is being assembled, the value contained in the location counter does not change. This value now indicates the location of the current data after boundary alignment.
5. After assembling the instruction, the assembler increments the location counter by the length of the assembled data. The location counter now holds the address of the next available location.

By default, the assembler assumes location counter 0 for each section. However, you can indicate the location counter you wish to use with the **.text**, **.data**, or **.lcomm** pseudo-ops for the text, data, and bss sections, respectively.

Chapter 4. 032 Microprocessor Instructions

CONTENTS

About This Chapter

The first part of this chapter defines the 032 Microprocessor instructions, shows the categories and formats of instructions, and explains the notational conventions used to describe instructions.

The second part of this chapter lists the instructions in alphabetical order by mnemonic. For each instruction, both the mnemonic and op code or op codes are shown.

If you already know the mnemonic of the instruction, just find the appropriate page in the instruction directory. If you don't know the name of the instruction you want, but you know what the instruction should do, then look up the instruction name by its category. (See "Categories of Instructions" on page 4-5.)

Notes:

1. The extended branch instructions are not listed in alphabetical order. Instead, they are listed under the branch instruction that has the same op code. For example, **beq** is listed under **bb**.
2. Appendix A lists all the instructions by op code and by mnemonic.

Introduction to Instructions

Instructions are statements that the assembler translates into a machine-readable form. The assembler converts mnemonics to op codes, and operands to sequences of binary numbers, so that the processor can perform some operation. Do not confuse assembler instructions with pseudo-ops. (See Chapter 5.)

In RT PC 032 Microprocessor assembler language, some mnemonics have a single associated op code. Other mnemonics have two op codes. In this case, the assembler examines the instruction's immediate data and chooses the appropriate op code.

Notes:

1. Do not use an unassigned op code. If the processor encounters one of these reserved op codes, a program check error occurs. For more information on program checks, see *Virtual Resource Manager Technical Reference.*
2. The processor does not support dynamic instruction modification. Any attempt by software to modify an instruction may result in unpredictable operation.

Categories of Instructions

Instructions are grouped into 11 classes:

- Storage access
- Address computation
- Branching
- Traps
- Moves and inserts
- Arithmetic
- Logical operations
- Shifts
- System Control Register manipulation
- Processor I/O
- System control.

These categories of instructions are discussed on the next few pages.

The system control instructions and some of the system control register instructions are privileged. Except as noted, you can use all the other instructions in unprivileged state.

Storage Access Instructions

The storage access instructions do not affect the Condition Status register. You can use all these instructions in unprivileged state.

l	Load
lc	Load Character
lh	Load Half
lha	Load Half Algebraic
lm	Load Multiple
st	Store
stc	Store Character
sth	Store Half
stm	Store Multiple
tsh	Test and Set Half

Storage accesses for halfwords ignore the low-order bit of the effective address. Storage accesses for words ignore the low-order two bits of the effective address. For example, suppose memory holds the following contents.

Hex Address	**Hex Contents**
@4000 0000	77
@4000 0001	88
@4000 0002	99
@4000 0003	AA
@4000 0004	BB
@4000 0005	CC
@4000 0006	DD

Note that addresses 4000 0000 and 4000 0004 are word boundaries. If you issue the instructions

```
cau  14, 0x4000(0)     # R14 = 0x4000 0000
l    15, 2(14)         # load a word from 0x4000 0002
                       # effective address is 0x4000 0002
```

then the address used is 0x4000 0000. The displacement of 2 is ignored, and GPR 15 is loaded with the hex value 778899AA.

With some instructions (including the long forms of instructions with short and long forms), you specify a 16-bit displacement. This displacement is sign-extended at runtime, then added to the contents of a base register to form an effective address.

Note: Specifying 0 as a base register tells the assembler to use the value 0 as a base; the assembler does not use the contents of register 0.

For the loads and stores, you can implicitly or explicitly specify a base register. (See Chapter 3.) The displacement **D2** can be absolute, relocatable, or external. If the label used in the **D2** field is relocatable but is not covered by a **.using**, the address of the label must be below 32K. If you want to implicitly specify a base register with **.using**, the displacement must be a label or an expression containing a label. (See the discussion of **D2** on page 4-24.)

It is possible to generate addresses that are reserved for system functions and are protected from access in the processor's unprivileged state. Attempting to access such an address causes a program check. See "Special Addresses" on page 3-7; also see *Virtual Resource Manager Technical Reference* for an explanation of the Program Check Status Word.

The only instructions that reference memory are the load and store instructions. Because the 032 Microprocessor is a pipelined processor, other instructions that manipulate data in registers may run at the same time as the load or store.

If a load or store fails (for example, because of a page fault or data address exception), the IAR for the failing operation is saved. The saved IAR thus contains the address of the failing instruction. After software handles the exception, 032 Microprocessor re-executes the load or store instruction.

If a data address exception occurs for the Load Multiple (**lm**) or Store Multiple (**stm**) instructions, however, several loads or stores can occur before the exception is detected. The processor does not restore GPRs or storage to the state that existed before the exception occurred. However, if **lm** causes an exception, the **lm** base address register is restored to its original value so that the **lm** instruction can be restarted.

Note that the processor is designed to optimize for full word operations. For example, the **lc, lh,** and **l** instructions all take up the same amount of machine time.

For the following storage access instructions, the assembler chooses a short or a long form:

l
lc
lha
st
stc
sth

The long forms have immediate data (the **D2** operand) 16 bits long. This immediate data represents a positive or negative displacement. Short forms have immediate data 4 bits long, representing a positive displacement.

When you specify the immediate data, the assembler automatically chooses the short form if the value of the displacement can be resolved during the assembler's first pass, and

- For **l** and **st**, D2 < 64 and **D2** is evenly divisible by 4
- For **lha** and **sth**, D2 < 32 and **D2** is evenly divisible by 2
- For **lc** and **stc**, D2 < 16.

Otherwise, the assembler chooses the long form of the instruction.

The **lh** instruction also has a long and a short form. In this case, the assembler chooses the short form if the **D2** operand is specified as zero. Otherwise, the assembler chooses the long form.

For more information about storage on RT PC, see Chapter 1. Also see the instructions in "Address Computation Instructions."

Address Computation Instructions

You can use all address computation instructions in unprivileged state.

cal6	Compute Address 16-Bit
cal	Compute Address Lower Half
call6	Compute Address Lower Half 16-Bit
cas	Compute Address Short
cau	Compute Address Upper Half
inc	Increment
dec	Decrement
lis	Load Immediate Short
lr	Load Register

The address computation instructions operate only on the contents of the general purpose registers. No storage references for operands occur. The resultant values are not inspected for address exceptions.

The contents of the Condition Status register are not changed by any of these instructions.

You can use some of these instructions for arithmetic, as well as for address computation. If you want to do arithmetic and set Condition Status bits, though, you should use the arithmetic instructions shown in "Arithmetic Instructions" on page 4-16.

Branch Instructions

You can use all branch instructions in unprivileged state.

b[x]	Branch [with Execute]
br[x]	Branch Using Register [with Execute]
bala[x]	Branch and Link Absolute [with Execute]
bali[x]	Branch and Link Immediate [with Execute]
balr[x]	Branch and Link Using Register [with Execute]
bb[x]	Branch on Condition Bit Immediate [with Execute]
bbr[x]	Branch on Condition Bit Using Register [with Execute]
bcc[x]	Branch on Carry Bit Clear [with Execute]
bccr[x]	Branch on Carry Bit Clear Using Register [with Execute]
bcs[x]	Branch on Carry Bit Set [with Execute]
bcsr[x]	Branch on Carry Bit Set Using Register [with Execute]
beq[x]	Branch on Equal [with Execute]
beqr[x]	Branch on Equal Using Register [with Execute]
bge[x]	Branch on Greater Than or Equal [with Execute]
bger[x]	Branch on Greater Than or Equal Using Register [with Execute]
bgt[x]	Branch on Greater Than [with Execute]
bgtr[x]	Branch on Greater Than Using Register [with Execute]
ble[x]	Branch on Less Than or Equal [with Execute]
bler[x]	Branch on Less Than or Equal Using Register [with Execute]
blt[x]	Branch on Less Than [with Execute]
bltr[x]	Branch on Less Than Using Register [with Execute]
bnb[x]	Branch on Not Condition Bit Immediate [with Execute]
bnbr[x]	Branch on Not Condition Bit Immediate Using Register [with Execute]
bne[x]	Branch on Not Equal [with Execute]
bner[x]	Branch on Not Equal Using Register [with Execute]
btc[x]	Branch on Test Bit Clear [with Execute]
btcr[x]	Branch on Test Bit Clear Using Register [with Execute]
bts[x]	Branch on Test Bit Set [with Execute]
btsr[x]	Branch on Test Bit Set Using Register [with Execute]
bvc[x]	Branch on Overflow Clear [with Execute]
bvcr[x]	Branch on Overflow Clear Using Register [with Execute]
bvs[x]	Branch on Overflow Set [with Execute]
bvsr[x]	Branch on Overflow Set Using Register [with Execute]

The branch instructions change the normal sequential execution of instructions. These instructions use different target addressing forms to provide subroutine linkage, decision making, and loop control.

The assembler allows branching within an assembler language section, but not between assembler language sections. In other words, the address of a branch target instruction may be external (outside the current file) or internal (within the current file) at assembly time. After linking, though, the branch reference must resolve to a symbol in the same segment as the branching instruction.

The **.using** and **.drop** pseudo-ops do not affect the branch instructions. Some branch instructions test bits in the Condition Status register, but branch instructions never set or clear any of these CS bits.

Kinds of Addresses in Branches

The branch instructions have three different kinds of branch target addresses: absolute immediate, absolute, and relative immediate.

- The absolute immediate branch instructions are **bala** and **balax**. These instructions have a 24-bit immediate operand. This operand, extended on the left with eight binary zeroes and with its low-order bit forced to zero, becomes the branch target address. These instructions are considered to be absolute because they get the branch value from immediate data, not from the Instruction Address Register.
- The absolute branch instructions are **bbr[x], bnbr[x]**, extended branch instructions with these op codes, and **balr[x]**. They have a register as an operand. The contents of this register become the branch target address. These instructions are considered to be absolute because their target address is determined by a specified register, not by the Instruction Address Register. These instructions are relocatable.
- Relative immediate instructions are **bb[x], bnb[x]**, the extended branch instructions with these op codes, and **bali[x]**. These instructions cause branching to occur relative to the Instruction Address Register, that is, relative to the address of the branch instruction itself. Relative immediate instructions allow an 8-bit or a 20-bit halfword offset between the branch instruction and the branch destination.

  The relative immediate branch instructions have an **A1** operand which must be a symbol. **A1** denotes a label which is at the address of the branch target instruction. When the instruction is assembled, the address of the branch instruction is subtracted from the address of **A1** and then algebraically shifted right one bit. The result is a number which represents the difference in halfwords between the addresses of the branch instruction and the target instruction. The assembler treats this number as if it were immediate data (an I field). A positive I denotes a forward branch; a negative I denotes a backwards branch.

  Note that the target of a relative immediate branch can be externally defined. In fact, the assembler does not check to see whether the target is defined at all. If you forget to define the target, the assembler will not give you any error messages.

See also "Addressing" on page 3-4.

Branch with Execute Instructions

Every branch instruction has a corresponding branch with execute instruction. The instruction immediately following a branch with execute is called the subject instruction. It is executed regardless of the branch decision, as if it preceded the branch. Using branch with execute instructions may enhance performance, since the subject instruction will be executed while the branch instruction is being processed.

The assembler assumes that the subject instruction is 32 bits long. If the subject instruction is 16 bits long, the assembler automatically inserts a two-byte **nop** (No Operation instruction), so that the subject is padded to 32 bits.

There are certain restrictions on the branch with execute instructions:

- The subject instruction cannot affect the branch decision. Any Condition Status changes caused by the subject instruction occur after the branch decision has been made.
- A branch with execute instruction and its subject instruction are considered to be a single instruction. Thus, interrupts are not honored between the execution of a branch with execute instruction and the execution of its subject instruction.
- Certain instructions are not allowed to be the subject of a branch with execute instruction. Since the branch with execute instructions change the normal sequential execution of instructions, the subject instruction cannot also change the instruction sequencing. (If it does, the processor may be put in an unpredictable state.) Therefore, **lps**, **svc**, and all branch and trap instructions cannot be subject instructions.

  In addition, for branch and link with execute instructions, the register containing the return address is available to the subject instruction. Therefore, the subject instruction must be constructed so that the return address is not unintentionally modified.

Subroutine Linkage (Branch and Link)

Subroutine linkage is provided by the branch and link instructions **bali[x]**, **bala[x]**, and **balr[x]**. These instructions cause a branch to a new instruction sequence but preserve a return address in an implicitly or explicitly designated general purpose register.

For the nonexecute forms of the instructions, the return address is the updated instruction address, which is the address of the halfword immediately following the branch and link instruction in storage.

For the execute forms of the instructions, the return address is the address of the halfword that is four bytes beyond the end of the branch and link with execute instruction (that is, the updated instruction address plus four). This allows four bytes following the branch and link with execute for the subject instruction. If the subject instruction requires only two bytes, the assembler automatically inserts a two-byte **nop** (No Operation instruction).

Conditional Branches

Decision making and loop control are provided by the conditional branch instructions **bb[x], bbr[x], bnb[x], bnbr[x]**, and other branch instructions with these op codes.

For conditional branch instructions **bb[x], bbr[x], bnb[x]**, and **bnbr[x]** only, the branch decision is based on any specified state of the rightmost eight bits (bits 24 through 31) of the Condition Status (CS). In this case, the value of the **I1** immediate data operand specifies the CS bit that is used for the branch decision. CS bit 24 is specified by an **I1** value of 0; CS bit 25 is specified by an **I1** value of 1; and so forth. For the long forms of these instructions, however, the object code representation of the CS bit is different from the value specified with the **I1** operand. Figure 4-1 shows the correspondence of CS bits to the assembler language source code and object code representations.

CS Bit # In SCR 15	**Name of Bit**	**Bit # Used in Branches (I1 Operand)**	**Bit # in Object Code (Long Form)**	**Bit # in Object Code (Short Form, bb and bnb Only)**
24	PZ	0	8	0
25	LT	1	9	1
26	EQ	2	10	2
27	GT	3	11	3
28	C0	4	12	4
29	--	--	--	--
30	OV	6	14	6
31	TB	7	15	7

Figure 4-1. Correspondence of CS bits to Branch Instructions

Extended Branch Instructions

Extended branch instructions are conditional branch instructions with the same op codes as **bb[x], bbr[x], bnb[x]**, and **bnbr[x]**. They implicitly designate the bit in the Condition Status register that should be tested. For example, **beq A** branches to A if the EQ bit in the CS register is one; this is equivalent to **bb 2,A**.

The directory later in this chapter does not list the extended branches in alphabetic order. Instead, each extended branch instruction is discussed along with its equivalent explicit branch instruction. For example, the extended branch instruction **beq** has the same op code as **bb**; the directory discusses **beq** at the **bb** instruction.

Figure 4-2 shows the relationships between the **bb** and **bnb** instructions and the extended branch instructions.

Name	Equivalent	Name	Equivalent
		b	bnb 0,
		bx	bnbx 0,
		br	bnbr 0,
		brx	bnbrx 0,
blt	bb 1,	bge	bnb 1,
bltx	bbx 1,	bgex	bnbx 1,
bltr	bbr 1,	bger	bnbr 1,
bltrx	bbrx 1,	bgerx	bnbrx 1,
beq	bb 2,	bne	bnb 2,
beqx	bbx 2,	bne	bnbx 2,
beqr	bbr 2,	bner	bnbr 2,
beqrx	bbrx 2,	bnerx	bnbrx 2,
bgt	bb 3,	ble	bnb 3,
bgtx	bbx 3,	blex	bnbx 3,
bgtr	bbr 3,	bler	bnbr 3,
bgtrx	bbrx 3,	blerx	bnbrx 3,
bcs	bb 4,	bcc	bnb 4,
bcsx	bbx 4,	bccx	bnbx 4,
bcsr	bbr 4,	bccr	bnbr 4,
bcsrx	bbrx 4,	bccrx	bnbrx 4,
bvs	bb 6,	bvc	bnb 6,
bvsx	bbx 6,	bvcx	bnbx 6,
bvsr	bbr 6,	bvcr	bnbr 6,
bvsrx	bbrx 6,	bvcrx	bnbrx 6,
bts	bb 7,	btc	bnb 7,
btsx	bbx 7,	btcx	bnbx 7,
btsr	bbr 7,	btcr	bnbr 7,
btsrx	bbrx 7,	btcrx	bnbrx 7,

Figure 4-2. Extended Branch Instructions

Variable-Length Branch Instructions

For the following branch instructions in the assembler text section, the assembler automatically chooses a short or long form of the instruction:

b
bb
bcc
bcs
beq
bge
bgt
ble
blt
bnb
bne
btc
bts
bvc
bvs

The operand representing the target must be a label, not an expression. The assembler generates a displacement in halfwords between the address of the branch instruction and the target address.

The assembler automatically chooses the short form of the instruction if:

- The value of the displacement can be resolved during the assembler's first pass, and
- The binary value of the displacement can be represented in eight bits or fewer.

In this case, the machine instruction will contain a JI field. The short form of a branch instruction is known internally to the assembler as a jump. You cannot specify a jump instruction, but the assembler generates jump instructions when your operand fits into an eight-bit immediate field. This allows a jump range of -127 to +128 halfwords from the jump instruction.

If the displacement is between 9 and 20 bits long, or if the value of the displacement cannot be determined at the assembler's first pass, the assembler automatically chooses the long form of the instruction. (The machine instruction will contain a BI field.)

At execution, BI or JI is shifted left one bit, so that the displacement in halfwords becomes a displacement in bytes.

Note: Branches in the assembler data section are always assembled in the long form.

Trap Instructions

You can use all trap instructions in unprivileged state.

tgte	Trap if Register Greater Than or Equal
ti	Trap on Condition Immediate
tlt	Trap if Register Less Than

The trap instructions are provided to test for a specified set of conditions. Programmers may define traps for events that should not occur (for example, an index out of range, or use of an invalid character).

If the conditions tested by a trap instruction are met, the program trap bit of the Program Check Status is set to one and a program check occurs. If the tested conditions are not met, instruction execution continues with the next sequential instruction.

The comparisons are performed on operands that are treated as 32-bit unsigned integers (logical quantities). The Condition Status is not changed by any of these instructions.

Move and Insert Instructions

You can use all move and insert instructions in unprivileged state.

mc03	Move Character Zero from Three
mc13	Move Character One from Three
mc23	Move Character Two from Three
mc33	Move Character Three from Three
mc30	Move Character Three from Zero
mc31	Move Character Three from One
mc32	Move Character Three from Two
mftb[i]	Move From Test Bit [Immediate]
mttb[i]	Move To Test Bit [Immediate]

This group of instructions is concerned with the movement of data between general purpose registers and between a general purpose register and the Test Bit of the Condition Status. None of these instructions alter the Condition Status unless data is moved into the test bit.

For **mftbi** and **mttbi**, the assembler examines immediate data supplied by the **I2** operand. If the immediate data is less than or equal to decimal 15, the assembler chooses the "upper half" form of the instruction. If the immediate data is greater than 15, the assembler chooses the "lower half" form of the instruction.

Arithmetic Instructions

You can use all arithmetic instructions in unprivileged state.

a[i]	Add [Immediate]
abs	Absolute
ae[i]	Add Extended [Immediate]
ais	Add Immediate Short
c[i]	Compare [Immediate]
cl[i]	Compare Logical [Immediate]
d	Divide Step
dec	Decrement
inc	Increment
m	Multiply Step
onec	One's Complement
s	Subtract
se	Subtract Extended
sf[i]	Subtract From [Immediate]
sis	Subtract Immediate Short
exts	Extend Sign
twoc	Two's Complement

The arithmetic operations treat the general purpose registers as 32-bit quantities in two's complement representation. Except for **inc** and **dec**, each of these instructions affects certain bits in the Condition Status register. However, the bits that are set, and the manner in which they are set, vary according to the instruction that is executed.

The arithmetic instructions affect the following bits in the Condition Status register:

- The LT Condition Status bit indicates the sign of a result. All the arithmetic instructions except Multiply Step, Divide Step, and the Compares set the LT bit to one if the sign bit of the result is one. The arithmetic compare instructions set the LT bit to one if the algebraic magnitude of a given operand is less than the algebraic magnitude of the other. The logical compare instructions set the LT bit to one if the unsigned magnitude of a given operand is less than the unsigned magnitude of the other. The Multiply Step and Divide Step instructions do not affect this bit.
- The EQ bit is set by all instructions except Multiply Step and Divide Step if the result is a field of 32 zeroes, or, in the case of the compare instructions, if the two comparands are equal. The Multiply Step and Divide Step instructions do not affect this bit.

- The GT bit indicates the sign of a non-zero result. All instructions except Multiply Step, Divide Step, and Compares set the GT bit to one if the sign bit of a non-zero result is zero. The arithmetic compare instructions set the GT bit if the algebraic magnitude of a given operand is greater than the algebraic magnitude of the other. The logical compare instructions set the GT bit to one if the unsigned magnitude of a given operand is greater than the unsigned magnitude of the other. The Multiply Step and Divide Step instructions do not affect this bit.
- The C0 bit indicates whether a carryout has occurred from bit 0. All instructions except Compares, Extend Sign, Divide Step, and Multiply Step set C0 to one if a carryout has occurred. The Extend Sign instruction does not affect C0. The Multiply Step and Divide Step instructions set C0 according to certain multiply and divide conditions. Add operations set C0 to one if a carry occurs and to zero if no carry occurs. Subtract operations set C0 to zero if a borrow occurs and to one if no borrow occurs.

  The extended instructions incorporate the state of the C0 bit into the result. The extended add instructions (**ae** and **aei**) cause the value of the C0 bit to be added to the sum of the two operands. In the Subtract Extended (**se**) instruction, the value of the first operand is added to the ones complement of the second operand, and the value of the C0 bit is added to the result.
- The OV bit indicates arithmetic overflow. All instructions except Extend Sign, Multiply Step, Divide Step, and Compares set the OV bit to one when the signed result of an operation cannot be represented by 32 bits. The Extend Sign and Multiply Step instructions do not affect this bit. The Divide Step instruction sets it according to a divide condition.

Note: The instructions at "Address Computation Instructions" on page 4-8 may also be used to do arithmetic. However, these instructions will not set or clear any Condition Status bits.

Logical Operation Instructions

You can use all logical instructions in unprivileged state.

clrb	Clear Bit
clz	Count Leading Zeroes
n	AND
nilo	AND Immediate Lower Half Extended Ones
nilz	AND Immediate Lower Half Extended Zeroes
niuo	AND Immediate Upper Half Extended Ones
niuz	AND Immediate Upper Half Extended Zeroes
o	OR
oil	OR Immediate Lower
oiu	OR Immediate Upper
setb	Set Bit
x	Exclusive OR
xil	Exclusive OR Immediate Lower Half
xiu	Exclusive OR Immediate Upper Half

The logical operations treat the contents of the general purpose registers as 32-bit unsigned integers. The exception is the Count Leading Zeroes (**clz**) instruction, which examines only the lower half of a register. All logical operations except **clz** set Condition Status bits LT, EQ, and GT according to the algebraic value expressed in two's complement representation. If the result is a negative value, LT is set to one; if it is zero, EQ is set to one; if it is positive and not zero, GT is set to one. The Condition Status is unaffected by **clz**.

For **clrb** and **setb**, the assembler examines immediate data supplied by the **I2** operand. If the immediate data is less than or equal to decimal 15, the assembler chooses the "upper half" form of the instruction. If the immediate data is greater than 15, the assembler chooses the "lower half" form of the instruction.

Shift Instructions

You can use all shift instructions in unprivileged state.

sar[i]	Shift Algebraic Right [Immediate]
sl[i]	Shift Left [Immediate]
slp[i]	Shift Left Paired [Immediate]
sr[i]	Shift Right [Immediate]
srp[i]	Shift Right Paired [Immediate]

Shift instructions operate on the content of a register or a register half. Immediate shifts specify a shift amount of 0 to 31 bits to the left or right based on the value of the immediate field. For these instructions, a shift amount greater than 31 bits results in a 32-bit shift. The non-immediate, indirect shifts specify a shift amount of 0 to 32 bits to the left or right based on the low-order six bits of a register.

All shifts set the Condition Status bits LT, EQ, and GT according to the algebraic value in the register after the shift is completed. All instructions except the Shift Algebraic Right instructions supply zeroes to the vacated bit positions.

The **slp[i]** and **srp[i]** instructions cause results to be placed in the pair of a designated register.

The immediate shift instructions have small and large forms:

sari **slpi** **srpi**
sli **sri**

For these instructions, the assembler examines immediate data supplied by the **I2** operand. If the immediate data is less than or equal to decimal 15, the assembler chooses the "small" form of the instruction. If the immediate data is greater than 15, the assembler chooses the "large" form of the instruction.

System Control Register Manipulation Instructions

The System Control Register instructions are privileged instructions. In privileged state, they can be used to change the value of any valid SCR. In unprivileged state, you can use the System Control Register instructions to change the Multiplier Quotient and Condition Status registers only.

mfs	Move from System Control Register
mts	Move to System Control Register
clrcb	Clear Bit in the Condition Status Register
setcb	Set Bit in the Condition Status Register

The System Control Register instructions provide a way to change data in the SCRs. All SCRs except the ICS (SCR 14) are dynamically changed by the processor. Therefore, if you put data into an SCR and then read the SCR, you will not necessarily get the same data that you put in. Also, if you use these instructions to modify reserved bits, you will get unpredictable results.

Processor I/O Instructions

Warning: Do not use these instructions in unprivileged state. Use of these instructions in unprivileged state will cause the machine to receive a hardware interrupt without precise indication of the address of the offending instruction.

ior	Input/Output Read
iow	Input/Output Write

Processor I/O (PIO) instructions are used to transfer data between the general purpose registers and system components.

The PIO instructions are themselves non-privileged. However, all devices accessible through these instructions are privileged, and therefore are only accessible from within the Virtual Resource Manager, if it exists. If you have AIX Operating System, you should use AIX Operating System kernel calls to perform I/O functions in unprivileged state.

For more information, see *Hardware Technical Reference.*

System Control Instructions

Warning: Do not use these instructions in unprivileged state.

lps	Load Program Status
svc	Supervisor Call
wait	Wait

These instructions control the execution state of the virtual machine (if it exists) by causing a trap to the Virtual Resource Manager (if it exists) or to the operating system.

For information about virtual machines, see *Virtual Resource Manager Technical Reference.*

Notational Conventions for Instructions

The chart below shows the notational conventions used to describe the instructions. "Source notation" shows the kinds of operands you write into source code. "Object notation" shows what the assembler puts into object code. If the source notation is the same as the object notation, then the value that you write into the assembler source code equals the value assembled into the object code.

All operations occur at runtime, unless otherwise noted.

Source Notation	Object Notation	Meaning
R1	R1	This is a general purpose register used as the first operand. R1 must be an integer, where $0 \leq R1 \leq 15$. R1 can be an expression; for simplicity's sake, the example instructions show R1 as a constant.
R2	R2	This is a general purpose register used as the second operand. R2 must be an integer, where $0 \leq R2 \leq 15$. R2 can be an expression; for simplicity's sake, the example instructions show R2 as a constant.
R3	R3	This is a general purpose register used as the third operand. R3 must be an integer, where $0 \leq R3 \leq 15$. R3 can be an expression; for simplicity's sake, the example instructions show R3 as a constant.
SCR1	SCR1	This is a System Control Register, always used as the first operand in certain instructions. SCR1 is an integer. In general, $0 \leq SCR1 \leq 15$, but some instructions further restrict the value of SCR1.

Source Notation	Object Notation	Meaning
I1	UI for **bala[x]**	This is twenty-four bits of immediate data, used as the first operand in a **bala** or **balax** instruction. In this case, I1 represents an absolute, non-relocatable address, so that UI=I1. UI can also be generated by an A1 operand.
	IE for branches	This is four bits of immediate data used as the first operand in some conditional branch instructions. I1 is the CS bit that should be tested. IE is the assembler's notation for this bit, where IE = I1+8. (See "Extended Branch Instructions" on page 4-12.)
	I1 for **svc** and **lps**	For **svc**, I1 is 16 bits of immediate data used as the first operand. For **lps**, I1 is one bit of immediate data used as the first operand.
I2	I2	This is immediate data used as the second operand. I2 can be an arithmetic constant or a symbol. The length of I2 depends on the instruction.
	IN	For **clrb, mftbi, mttbr, setb** only, $0 \leq I2 \leq 31$, and ● if I2 ≤ 15, then IN = I2 (instruction affects upper half of register) ● if I2 > 15, then IN = I2-16 (instruction affects lower half of register). For **sari, sri, srpi, sli, slpi** only, $0 \leq I2 \leq 31$, and ● if I2 ≤ 15, then IN = I2 (small shift) ● if I2 > 15, then IN = I2-16 (large shift).
I3	I3	This is immediate data used as the third operand. I3 can be an arithmetic constant or a symbol. The length of I3 depends on the instruction.

Source Notation	Object Notation	Meaning
A1		This is a label used as the first operand in some branch instructions. A1 is the address of the target instruction (that is, the branch destination). The assembler subtracts the address of A1 from the address of the branch instruction and divides by two to get a displacement in halfwords.
	JI	The assembler chooses the short form of the branch instruction if the number of halfwords displacement fits in eight or fewer bits, *and* if it can be determined in the assembler's first pass. The assembler places the number of halfwords displacement into the JI field of the object code. The relative displacement is thus embedded in the object code just as if it were immediate data.
	BI	The assembler chooses the long form of the branch instruction if the number of halfwords displacement fits in nine to twenty bits, *or* if it cannot be determined at the assembler's first pass. The assembler places the number of halfwords displacement into the BI field of the object code. The relative displacement is thus embedded in the object code just as if it were immediate data.
	UI for **bala[x]**	This is a label representing an absolute branch destination, used as the only operand in a **bala** or **balax** instruction. The assembler puts the lowest 24 bits of this address into the UI field of the object code. UI can also be generated by an I1 operand.
(R1) *or* (R2) *or* (R3)	varies	This is the content of the designated register. When you type this operand, you type the parentheses as well. You may omit this operand if the register is currently specified by a **.using** pseudo-op. (If the register is covered by a **.using**, you can override the **.using** by explicitly specifying another register.)
0/(R2)	varies	This means you should either type a register number in parentheses, or omit the operand altogether. If R2 is not 0, this indicates the content of register R2. If R2 is 0, or if R2 is not specified at all, this indicates that the operand is omitted. The assembler uses the value 0 for a base.

Source Notation	Object Notation	Meaning
D2(R2)	varies	This is an explicit based address. R2 is a base register containing a base address, and D2 is the displacement from the base address. At runtime, D2 is added to the contents of R2 to form an effective address. D2 has a value that fits into 4 bits or 16 bits, depending on the instruction. D2 can be immediate data or a label. In general, D2 can be an external, relocatable, or absolute expression, but some instructions require D2 to be a certain type of expression. D2 can be an arithmetic constant only if you are not covering D2 with a **.using**. If you want to use **.using**, D2 must be a label or an expression involving a label. If the label representing a D2 operand is covered by a **.using R2**, you do not have to specify the (R2) part of the operand. The assembler will fill in the R2 value for you. D2 is assembled into the object code as a displacement. If D2 is a relocatable expression, the value of D2 could change at load time. For certain instructions, you can use the **.xaddr** pseudo-op to get displacements larger than 15 bits. See **.xaddr** in Chapter 5.
//	varies	This is a concatenation of the fields specified on either side of the bars. For example, *4 zeroes//I2* means a value of *0000I2*. *I2//4 zeroes* means a value of *I20000.*
@	none	This is the hex address of an instruction. In the examples, an instruction's address is shown to the left of the instruction. You will not see this address in your program, and the address is not part of the instruction. These sample addresses are shown only to help you understand where branches go.

Instruction Formats

The seven instruction formats are shown in Figure 4-3.

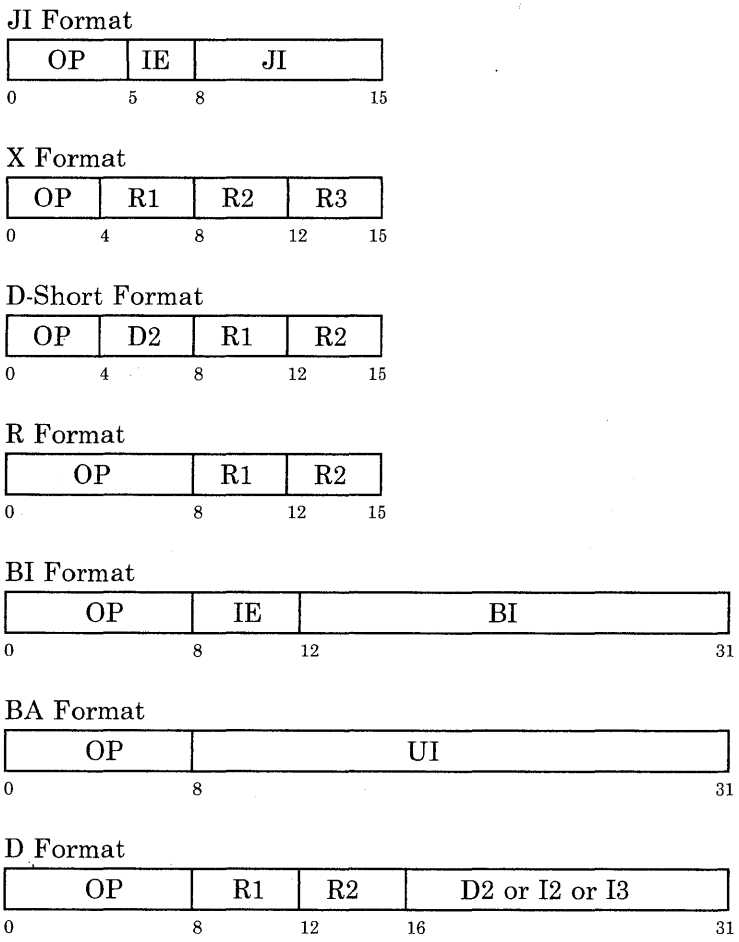


Figure 4-3. Instruction Formats

As Figure 4-4 shows, instructions are either two or four bytes in length. The first four, five, or eight bits of an instruction are referred to as the op code.

Format	Length of Op Code	Length of Instruction
JI	5 bits	2 bytes
X	4 bits	2 bytes
D-short	4 bits	2 bytes
R	8 bits	2 bytes
BI	8 bits	4 bytes
BA	8 bits	4 bytes
D	8 bits	4 bytes

Figure 4-4. Summary of Instruction Formats

For X, D-Short, and D format instructions that refer to main storage or system components, the address is calculated according to the following formulas:

X Format	(R2) + 0/(R3)
D-Short Format	0/(R2) + 28 zeroes//D2 0/(R2) + 27 zeroes//D2//1 zero 0/(R2) + 26 zeroes//D2//2 zeroes
D Format	0/(R2) + 16 zeroes//I3 0/(R2) + 16 zeroes//D2 0/(R2) + Sign-Extended I3 0/(R2) + Sign-Extended D2

Directory of Instructions

This directory lists the instructions in alphabetic order by mnemonic. You should use this directory if you know the mnemonic of the instruction you need. If you do not know the name of the instruction you need, refer to "Categories of Instructions" on page 4-5.

The directory entry for each instruction includes a purpose, format, and example. Some directory entries also have remarks about the instruction.

Notes:

1. The Format section of each directory entry does not show labels being used with instructions. However, programmers may place a label or labels in front of any instruction.
2. Many example instructions assume that a register has a certain value. Also assume that this value does not change for the rest of the example, unless otherwise noted.
3. All 32-bit hex values are shown with a space between halfwords. This space does not represent the way data is stored in registers or in memory. For example, in 0x0123 4567, the space between hex digits 3 and 4 is shown only to make the number easier to read.
4. All the mnemonics are listed in alphabetic order except for the extended branches. Extended branch instructions are discussed at their parent instruction. For example, the **bvs** extended branch is discussed at the **bb** mnemonic.

Add a

Purpose: The contents of registers R1 and R2 are added. The result is placed into register R1.

Format: a R1,R2

E1	R1	R2
0	8	12 15

Remarks: Condition Status bits LT, EQ, GT, C0 and OV are affected.

Example:

```
  # assume GPR 4 holds 0x9000 3000
  # assume GPR 10 holds 0x8000 7000
a 4,10
  # now GPR 4 holds 0x1000 A000
  # OV, CO, and GT bits are set to 1
```

See Also: "Arithmetic Instructions" on page 4-16

abs Absolute

Purpose: The content of register R1 is replaced by the absolute value of the content of register R2.

Format: abs R1,R2

E0	R1	R2
0	8	12 15

Remarks:

- Condition Status bits LT, EQ, GT, C0 and OV are affected. Normally, only Condition Status bits EQ or GT are set to one according to the result; the remaining affected bits are set to zero.
- If register R2 contains the maximum negative number for which there is no equivalent positive number (-2^{31}), then the content of register R1 is set equal to the content of register R2, and the Condition Status bits LT and OV are set to one.

Examples:

```
  # assume GPR 10 holds 0x7000 3000
abs 4,10
  # now GPR 4 holds 0x7000 3000
  # GT bit is set to 1


  # assume GPR 6 holds 0xFFFF FFFF
abs 5,6
  # now GPR 5 holds 0x0000 0001
  # GT bit is set to 1
```

See Also: "Arithmetic Instructions" on page 4-16

Add Extended ae

Purpose: The content of register R1, the content of register R2, and the value of Condition Status bit C0 are summed. The result is placed into register R1.

Format: ae R1,R2

F1	R1	R2
0	8	12 15

Remarks: Condition Status bits LT, EQ, GT, OV, and C0 are affected. If 32 bits are not sufficient to hold the signed sum of the two GPRs and CS bit C0, then CS bit OV is set. This simulates a carry-over out of the high-order bit and allows software to support multiple precision addition by testing bit OV.

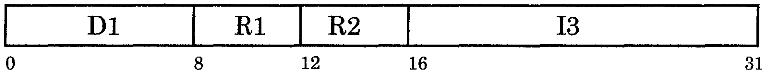
Examples:

```
  # assume bit C0 is on

  # assume GPR 4 holds 0x1000 0400
  # assume GPR 10 holds 0x1000 6008
ae 4,10
  # now GPR 4 holds 0x2000 6409
  # GT bit is set to one

  # assume GPR 6 holds 0x1000 0400
  # assume GPR 10 holds 0xEFFF FFFF
ae 6,10
  # now GPR 6 holds 0x0000 0400
  # GT and C0 bits are set to one
```

See Also: "Arithmetic Instructions" on page 4-16

Purpose: The field I3 is sign extended. The sign-extended I3, the content of register R2, and the value of Condition Status bit C0 are then summed. The result is placed in register R1.

Format: aei R1,R2,I3



Remarks:

- Condition Status bits LT, EQ, GT, C0, and OV are affected.
- This allows multiple precision addition.

Examples:

```
      # assume GPR 4 holds 0x0000 0332
      # assume CS bit CO is on

aei 3,4,0x4E

      # GPR 3 now holds 0x0000 0381
      # GT bit is set to one

aei 3,4,0xCE00

      # sign-extended 0xCE00 is 0xFFFF CE00
      # GPR 3 now holds 0xFFFF D133
      # LT bit is set to one
```

See Also: "Arithmetic Instructions" on page 4-16

Add Immediate   ai

Purpose: The field I3 is sign extended, then added to the content of register R2. The result is placed in register R1.

Format: ai R1,R2,I3

C1	R1	R2	I3	
0	8	12	16	31

Remarks: Condition Status bits LT, EQ, GT, OV and C0 are affected. Bit C0 turns on if the sign-extended I3 plus the content of register R2 results in a carry out of bit zero.

Example:

```
  # assume GPR 10 holds 0x0000 2346
ai 9,10,0xFFFF
  # sign-extended I3 becomes 0xFFFF FFFF
  # GPR 9 now holds 0x0000 2345
  # GT and C0 bits are set to one
```

See Also: “Arithmetic Instructions” on page 4-16

Purpose: I2 is extended on the left with 28 zeroes, then added to the content of register R1. The result is placed in register R1.

Format: ais R1,I2

90	R1	I2
0	8	12 15

Remarks:

- I2 must have a value between decimal 0 and 15 inclusive.
- Condition Status bits LT, EQ, GT, OV and C0 are affected.

Examples:

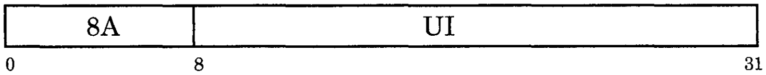
```
   # assume GPR 4 holds 0x1100 44CC
ais 4,0xE
   # GPR 4 now holds 0x110044DA
   # GT bit is set to one

   # assume GPR 5 holds 0xFFFF FFFC
ais 5,6
   # GPR 5 now holds 0x0000 0002
   # GT and C0 bits are set to one
```

See Also: "Arithmetic Instructions" on page 4-16

Purpose: The content of GPR 15 is replaced by the updated instruction address. The updated instruction address is replaced by eight zeroes//UI, with its rightmost bit forced to zero.

If the operand is 24-bit immediate data (I1), UI equals I1. If the operand is a label (A1), UI is the low-order 24 bits of the address of A1.

Format: bala A1 or bala I1



Remarks:

- If immediate data is used as an operand, the instruction will not be relocatable.
- This instruction may be useful to kernel programmers who want to access the lowest 32K of memory. Chapter 3 lists invalid storage locations (invalid targets of a **bala**). Chapter 5 discusses the **.direct** pseudo-op, which allows direct addressing.

Examples:

Example 1

```
        @1000 5000     bala 0x004501
                        # GPR 15 now holds 0x1000 5004
                        # updated IAR now holds 0x0000 4500
        @1000 5004    back:
```

Example 2

```
        @0000 4500    here:
                          .
                          .
        @2000 5000     bala here
                        # GPR 15 now holds 0x1000 5004
                        # updated IAR now holds 0x0000 4500
        @2000 5004    return:
```

See Also: "Branch Instructions" on page 4-9

Branch and Link Absolute with Execute **balax**

Purpose: The content of GPR 15 is replaced by the updated instruction address incremented by four. The updated instruction address is replaced by eight zeroes//UI with its rightmost bit forced to zero. The instruction immediately following the branch instruction is executed before the target instruction is executed.

If the operand is 24-bit immediate data (I1), UI equals I1. If the operand is a label (A1), UI is the low-order 24 bits of the address of A1.

Format: balax A1 or balax I1

8B	UI
0	8 31

Remarks:

- If immediate data is used as an operand, the instruction will not be relocatable.
- This instruction may be useful to kernel programmers who want to access the lowest 32K of memory. Chapter 3 lists invalid storage locations (invalid targets of a **balax**). Chapter 5 discusses the **.direct** pseudo-op, which allows direct addressing.

Example:

```
@1000 5000              balax 0x004501
@1000 5004   first:     ais 6,7
              # this instruction executes first,
              # then IAR is updated to 0x0000 4500
              # GPR 15 now contains 0x1000 5008
              # instruction at @0000 4500 executes next
@1000 5008   back:
```

See Also: "Branch Instructions" on page 4-9

Purpose: The content of register R1 is replaced by the updated instruction address. The assembler subtracts the address of the branch instruction from the address of A2. The assembler then divides the result by two to form BI, the displacement in halfwords. At execution, BI is sign extended and shifted left one bit to form the displacement in bytes. This byte displacement is added to the address of the branch instruction. The result then replaces the updated instruction address.

Format: bali R1,A2

8C	R1	BI	
0	8	12	31

Example:

```
@1000 5000   here:  bali 15,there
             # GPR 15 holds 0x1000 5004
             # BI is 0x30
             # updated IAR now 0x1000 5060
             # branch occurs to there
@1000 5004   back:
            .
            .
@1000 5060   there:
```

See Also: "Branch Instructions" on page 4-9

Branch and Link Immediate with Execute balix

Purpose: The instruction immediately following the branch instruction is executed.

The content of register R1 is replaced by the updated instruction address incremented by four. The assembler subtracts the address of the branch instruction from the address of A2. The assembler then divides the result by two to obtain BI, the displacement in halfwords. At execution, BI is sign extended and shifted left one bit to form the displacement in bytes. This byte displacement is added to the branch instruction address. The result then replaces the updated instruction address.

Format: balix R1,A2

8D	R1	BI
0	8	12 31

Example:

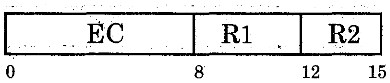
```
@1000 5000   here:   balix 15,there
            # BI is 0x30
            # updated IAR now holds 0x1000 5060
@1000 5004          lis 5,0xF
            # this executes
            # GPR 15 holds 0x1000 5008
            # now branch occurs to there
@1000 5008   back:
            .
            .
@1000 5060   there:
```

See Also: "Branch Instructions" on page 4-9

balr — Branch and Link Using Register

Purpose: The content of register R1 is replaced by the updated instruction address. The updated instruction address is replaced by the content of register R2 with the rightmost bit forced to zero.

Format: balr R1,R2



Example:

```
                # assume GPR 12 holds 0x1080 0100
@1080 0000      here:  balr 5,12
                # GPR 5 now holds 0x1080 0004
                # IAR now holds 0x1080 0100
                # branch occurs to there
@1080 0004      back:
               .
               .
@1080 0100      there:
```

See Also: “Branch Instructions” on page 4-9

Branch and Link Using Register with Execute balrx

Purpose: The instruction immediately following the branch instruction is executed.

The content of register R1 is replaced by the updated instruction address incremented by four. The updated instruction address is replaced by the content of register R2 with the rightmost bit forced to zero.

Format: balrx R1,R2

ED	R1	R2
0	8	12 15

Example:

```
          # assume GPR 12 holds 0x1080 0100
@1080 0000  here:   balrx 5,12
@1080 0004          lis  6,0xF

          # the lis executes
          # GPR 5 now holds @1080 0008
          # IAR now holds 0x1080 0100
          # branch occurs to there

@1080 0008     back:
               .
               .
@1080 0100     there:
```

See Also: "Branch Instructions" on page 4-9

bcs, beq, bgt, blt, bts, bvs Extended Branches also

Purpose: For the **bb** instruction, you explicitly specify a Condition Status bit to be tested. The assembler subtracts the address of the branch instruction from the address of A2. The assembler then divides by two to obtain BI or JI, the displacement in halfwords. If the Condition Status bit specified by I1 is one at execution, BI or JI is sign extended and shifted left one bit to form the displacement in bytes. This byte displacement is added to the branch instruction address. The result then replaces the updated instruction address.

The extended branch instructions based on **bb** are:

blt	Branch on Less Than
beq	Branch on Equal
bgt	Branch on Greater Than
bcs	Branch on Carry Bit Set
bvs	Branch on Overflow Set
bts	Branch on Test Bit Set.

Each of these instructions implicitly tests a CS bit, so that you do not have to specify the bit to test. For example, **blt** is equivalent to **bb 1**. Figure 4-2 on page 4-13 shows the correspondence of the extended branches to the **bb** instruction.

For the extended branches, the assembler subtracts the address of the branch instruction from the address of A1. The assembler then divides the result by two to obtain BI or JI, the displacement in halfwords. If the implicitly specified Condition Status bit is one at execution, BI or JI is sign extended and shifted left one bit to form the displacement in bytes. This byte displacement is added to the branch instruction address. The result then replaces the updated instruction address.

Format:

bb I1,A2 blt A1 beq A1 bgt A1
bcs A1 bvs A1 bts A1

long form

8E	IE	BI
0	8	12 31

short form

0	1	IE	JI
0	4	5	8 15

Remarks:

- If the implicitly or explicitly specified Condition Status bit is zero, the updated instruction address is unaltered.
- For **bb**, I1 must have an integer value of 0, 1, 2, 3, 4, 6, or 7.
- For the long form of **bb**, IE = I1+8. For the short form of **bb**, IE = I1. (See Figure 4-1 on page 4-12.) For the extended branches, there is no I1 operand; the assembler inserts the proper IE value into the object code.
- A **bb 0** would be a no-operation instruction, since the PZ bit is always zero. However, you should not use a no-operation to make software timing loops. Instead, you should rely on the system timer (**mts** and **mfs**) or real time clock (**ior** and **iow**) facilities if you must write timing-dependent code.

Examples: *Example 1*

```
                     setcb 15,11
@1030 0100   pan:    bb 3,fire
          # displacement between pan and fire is 0x200 halfwords
          # op code 8E is generated
          # BI is 0x00200
          # branch occurs to fire

              .
              .
@1030 0500  fire:
```

Example 2

```
                      setcb 15,10
@2000 0100   here:    beq there
          # displacement between here and there is 0x200 halfwords
          # op code 8E is generated
          # BI is 0x00200
          # branch occurs to there

              .
              .
@2000 0500  there:
```

Example 3

```
                       setcb 15,10
@3000 0100   order:    bb 2,chaos
          # displacement between order and chaos is 6 halfwords
          # op code 0 is generated
          # JI is 0x06
          # branch occurs to chaos

              .
              .
@3000 010C  chaos:
```

Example 4

```
                      setcb 15,10
@2000 0100   dark:    beq light
          # displacement between dark and light is 6 halfwords
          # op code 0 is generated
          # JI is 0x06
          # branch occurs to light

              .
              .
@2000 010C  light:
```

See Also: "Branch Instructions" on page 4-9

setcb on page 4-132

bbr — Branch on Condition Bit Using Register

bcsr, beqr, bgtr, bltr, btsr, bvsr Extended Branches also

Purpose: For the **bbr** instruction, you explicitly specify a Condition Status bit to be tested. If the Condition Status bit specified by I1 is one, the updated instruction address is replaced by the content of register R2 with the rightmost bit forced to zero.

The extended branch instructions based on **bbr** are:

bltr	Branch on Less Than Using Register
beqr	Branch on Equal Using Register
bgtr	Branch on Greater Than Using Register
bcsr	Branch on Carry Bit Set Using Register
bvsr	Branch on Overflow Set Using Register
btsr	Branch on Test Bit Set Using Register.

Each of these instructions implicitly tests a CS bit, so that you do not have to specify the bit to test. For example, **bltr** is equivalent to **bbr 1**. Figure 4-2 on page 4-13 explains the correspondence of the extended branches to the **bbr** instruction.

For the extended branches, if the implicitly specified Condition Status bit is one, the updated instruction address is replaced by the content of register R1 with the rightmost bit forced to zero.

Format:

bbr I1,R2 bltr R1 beqr R1 bgtr R1
bcsr R1 bvsr R1 btsr R1

for **bbr**

EE	IE	R2
0	8	12 15

for extended branches

EE	IE	R1
0	8	12 15

Remarks:

- If the implicitly or explicitly specified Condition Status bit is zero, the updated instruction address is unaltered.
- For **bbr**, I1 must have an integer value of 0, 1, 2, 3, 4, 6, or 7.
- For **bbr**, IE = I1+8. For the extended branches, there is no I1 operand; the assembler inserts the proper IE value into the object code.

Examples: *Example 1*

```
                        # assume GPR 12 holds 0x2080 0100

                           setcb 15,10
               here:       bbr 2,12

                        # updated IAR now 0x2080 0100
                        # branch occurs to there
                    .
                    .
@2080 0100     there:
```

Example 2

```
                        #assume GPR 12 holds 0x3080 0100

                           setcb 15,10
               pan:        beqr 12

                        # updated IAR now 0x3080 0100
                        # branch occurs to fire
                    .
                    .
@3080 0100     fire:
```

See Also: "Branch Instructions" on page 4-9

setcb on page 4-132

bbrx Branch on Condition Bit Using Register with Execute

bcsrx, beqrx, bgtrx, bltrx, btsrx, bvsrx Extended Branches also

Purpose: For the **bbrx** instruction, you explicitly specify a Condition Status bit to be tested. If the Condition Status bit specified by I1 is one, the updated instruction address is replaced by the content of register R2 with the rightmost bit forced to zero. The instruction immediately following the branch instruction is executed before the target instruction is executed.

The extended branch instructions based on **bbrx** are:

bltrx	Branch on Less Than Using Register with Execute
beqrx	Branch on Equal Using Register with Execute
bgtrx	Branch on Greater Than Using Register with Execute
bcsrx	Branch on Carry Bit Set Using Register with Execute
bvsrx	Branch on Overflow Set Using Register with Execute
btsrx	Branch on Test Bit Set Using Register with Execute.

Each of these instructions implicitly tests a CS bit, so that you do not have to specify the bit to test. For example, **bltrx** is equivalent to **bbrx 1**. Figure 4-2 on page 4-13 explains the correspondence between **bbrx** and the extended branches.

For the extended branches, if the implicitly specified Condition Status bit is one, the updated instruction address is replaced by the content of register R1 with the rightmost bit forced to zero. The instruction immediately following the branch instruction is executed before the target instruction is executed.

Format:

```
bbrx I1,R2   bltrx R1   beqrx R1   bgtrx R1
bcsrx R1     bvsrx R1   btsrx R1
```

for **bbrx**

EF	IE	R2
0	8	12 15

for extended branches

EF	IE	R1
0	8	12 15

Remarks:

- If the implicitly or explicitly specified Condition Status bit is zero, the updated instruction address is unaltered.
- For **bbrx**, I1 must have an integer value of 0, 1, 2, 3, 4, 6, or 7.
- For **bbrx**, IE = I1+8. For the extended branches, there is no I1 operand; the assembler inserts the proper IE value into the object code.

Examples: *Example 1*

```
             # assume GPR 12 holds 0x2080 0100

                      setcb 15,10
@1800 0000   here:    bbrx 2,12
@1800 0004            clrb 6,13

          # first clrb executes
          # updated IAR now 0x2080 0100
          # branch occurs to there

             .
             .
@2080 0100   there:
```

Example 2

```
                  # assume GPR 12 holds 0x4080 0100

                           setcb 15,10
  @3800 0000      pan:     beqrx 12
  @3800 0004               clrb 6,13

               # first clrb executes
               # updated IAR now 0x4080 0100
               # branch occurs to fire

                  .
                  .
  @4080 0100      fire:
```

See Also: "Branch Instructions" on page 4-9

Branch on Condition Bit Immediate with Execute bbx

bcsx, beqx, bgtx, bltx, btsx, bvsx Extended Branches also

Purpose: For the **bbx** instruction, you explicitly specify a Condition Status bit to be tested. The assembler subtracts the address of the branch instruction from the address of A2. The assembler then divides the result by two to obtain BI, the displacement in halfwords. If the Condition Status bit specified by I1 is one at execution, BI is sign extended and shifted left one bit to form the displacement in bytes. This byte displacement is added to the branch instruction address. The result then replaces the updated instruction address. The instruction immediately following the branch instruction is executed before the target instruction is executed.

The extended branch instructions based on **bbx** are:

bltx	Branch on Less Than with Execute
beqx	Branch on Equal with Execute
bgtx	Branch on Greater Than with Execute
bcsx	Branch on Carry Bit Set with Execute
bvsx	Branch on Overflow Set with Execute
btsx	Branch on Test Bit Set with Execute.

Each of these instructions implicitly tests a CS bit, so that you do not have to specify the bit to test. For example, **bltx** is equivalent to **bbx 1**. Figure 4-2 on page 4-13 explains the correspondence of the **bbx** instruction with the extended branches.

For the extended branches, the assembler subtracts the address of the branch instruction from the address of A1. The assembler then divides the result by two to obtain BI, the displacement in halfwords. If the implicitly specified Condition Status bit is one at execution, BI is sign extended and shifted left one bit to form the displacement in bytes. This byte displacement is added to the branch instruction address. The result then replaces the updated instruction address. The instruction immediately following the branch instruction is executed before the target instruction is executed.

Format:

bbx I1,A2 bltx A1 beqx A1 bgtx A1
bcsx A1 bvsx A1 btsx A1

8F	IE	BI	
0	8	12	31

Remarks:

- If the implicitly or explicitly specified Condition Status bit is zero, the updated instruction address is unaltered.
- For **bbx**, I1 must have an integer value of 0, 1, 2, 3, 4, 6, or 7.
- For **bbx**, IE = I1+8. (See Figure 4-1 on page 4-12.) For the extended branches, there is no I1 operand; the assembler inserts the proper IE value into the object code.

Examples:

Example 1

```
                   setcb 15,10
@1000 0100  here: bbx 2,there

                   #  displacement between here and there is 0x200 halfwords
                   #  BI is 0x00200

@1000 0104         clrb 6,13

                   #  at execution, this instruction executes
                   #  updated IAR replaced by 0x1000 0500
                   #  branch occurs to there

                   .
                   .
@1000 0500  there:
```

Example 2

```
                  setcb 15,10
@3000 0100  pan:  beqx fire

                  #  displacement between pan and fire is 0x200 halfwords
                  #  BI is 0x00200

@3000 0104        clrb 6,13

                  #  at execution, this instruction executes
                  #  updated IAR replaced by 0x3000 0500
                  #  branch occurs to fire

                  .
                  .
@3000 0500  fire: .
```

See Also: "Branch Instructions" on page 4-9

bnb **Branch on Not Condition Bit Immediate**

b, bcc, bge, ble, bne, btc, bvc Extended Branches also

Purpose: For the **bnb** instruction, you explicitly specify a Condition Status bit to be tested. The assembler subtracts the address of the branch instruction from the address of A2. The assembler then divides the result by two to obtain BI or JI, the displacement in halfwords. If the Condition Status bit specified by I1 is zero at execution, BI or JI is sign extended and shifted left one bit to form the displacement in bytes. This byte displacement is added to the branch instruction address. The result then replaces the updated instruction address.

The extended branch instructions based on **bnb** are:

b	Branch
bge	Branch on Greater Than or Equal
bne	Branch on Not Equal
ble	Branch on Less Than or Equal
bcc	Branch on Carry Bit Clear
bvc	Branch on Overflow Clear
btc	Branch on Test Bit Clear.

Each of these instructions implicitly tests a CS bit, so that you do not have to specify the bit to test. For example, **b** is equivalent to **bnb 0**. Figure 4-2 on page 4-13 explains the correspondence of **bnb** with the extended branches.

For the extended branches, the assembler subtracts the address of the branch instruction from the address of A1. The assembler then divides the result by two to obtain BI or JI, the displacement in halfwords. If the implicitly specified Condition Status bit is zero at execution, BI or JI is sign extended and shifted left one bit to form the displacement in bytes. This byte displacement is added to the branch instruction address. The result then replaces the updated instruction address.

Format:

bnb I1,A2	b A1	bge A1	bne A1
ble A1	bcc A1	bvc A1	btc A1

long form

88	IE	BI	
0	8	12	31

short form

0	IE	JI	
0	5	8	15

Remarks:

- If the implicitly or explicitly specified Condition Status bit is one, the updated instruction address is unaltered.
- For **bnb**, I1 must have an integer value of 0, 1, 2, 3, 4, 6, or 7.
- For the long form of **bnb**, IE = I1 + 8. For the short form of **bnb**, IE = I1. (See Figure 4-1 on page 4-12.) For the extended branches, there is no I1 operand; the assembler inserts the proper IE value into the object code.
- Note that **b** always branches, since the PZ bit is always zero.

Examples: *Example 1*

```
                 clrcb 15,11
@1030 0100  pan: bnb 3,fire
            # displacement between pan and fire is 0x200 halfwords
            # op code 88 is generated
            # BI is 0x00200
            # branch occurs to fire
            .
            .
@1030 0500  fire:
```

Example 2

```
                 clrcb 15,10
@2000 0100  here: bne there
            # displacement between here and there is 0x200 halfwords
            # op code 88 is generated
            # BI is 0x00200
            # branch occurs to there
            .
            .
@2000 0500  there:
```

Example 3

```
                     clrcb 15,10
@3000 0100   order:    bnb 2,chaos
          # displacement between order and chaos is 6 halfwords
          # op code 0 is generated
          # JI is 0x06
          # branch occurs to chaos

              .
              .
@3000 010C  chaos:
```

Example 4

```
                    clrcb 15,10
@4000 0100   dark:    bne light
          # displacement between dark and light is 6 halfwords
          # op code 0 is generated
          # JI is 0x06
          # branch occurs to light

              .
              .
@4000 010C  light:
```

See Also: "Branch Instructions" on page 4-9

clrcb on page 4-79

bnbr Branch on Not Condition Bit Using Register

br, bccr, bger, bler, bner, btcr, bvcr Extended Branches also

Purpose: For the **bnbr** instruction, you explicitly specify a Condition Status bit to be tested. If the Condition Status bit specified by I1 is zero at execution, the updated instruction address is replaced by the content of register R2 with the rightmost bit forced to zero.

The extended branch instructions based on **bnbr** are:

br	Branch Using Register
bger	Branch on Greater Than or Equal Using Register
bner	Branch on Not Equal Using Register
bler	Branch on Less Than or Equal Using Register
bccr	Branch on Carry Bit Clear Using Register
bvcr	Branch on Overflow Clear Using Register
btcr	Branch on Test Bit Clear Using Register.

Each of these instructions implicitly tests a CS bit, so that you do not have to specify the bit to test. For example, **br** is equivalent to **bnbr 0**. Figure 4-2 on page 4-13 explains the correspondence between **bbr** and the extended branches.

For the extended branches, if the implicitly specified Condition Status bit is zero at execution, the updated instruction address is replaced by the content of register R1 with the rightmost bit forced to zero.

Format:

bnbr I1,R2	br R1	bger R1	bner R1
bler R1	bccr R1	bvcr R1	btcr R1

for **bnbr**

E8	IE	R2
0	8	12 15

for extended branches

E8	IE	R1
0	8	12 15

Remarks:

- If the implicitly or explicitly specified Condition Status bit is one, the updated instruction address is unaltered.
- For **bnbr**, I1 must have an integer value of 0, 1, 2, 3, 4, 6, or 7.
- For **bnbr**, IE = I1+8. For the extended branches, there is no I1 operand; the assembler inserts the proper IE value into the object code.
- Note that **br** always branches, since the PZ bit is always zero.

Examples: *Example 1*

```
                # assume GPR 12 holds 0x2080 0100

                       clrcb 15,10
                  here: bnbr 2,12

                # updated IAR now 0x2080 0100
                # branch occurs to there

                    .
                    .
     @2080 0100   there:
```

Example 2

```
                # assume GPR 12 holds 0x3080 0100

                      clrcb 15,10
                  pan: bner 12

                # updated IAR now 0x3080 0100
                # branch occurs to fire

                    .
                    .
     @3080 0100   fire:
```

See Also: "Branch Instructions" on page 4-9

clrcb on page 4-79

Branch on Not Condition Bit Using Register with Execute **bnbrx**

brx, bccrx, bgerx, blerx, bnerx, btcrx, bvcrx Extended Branches also

Purpose: For the **bnbrx** instruction, you explicitly specify a Condition Status bit to be tested. If the Condition Status bit specified by I1 is zero at execution, the updated instruction address is replaced by the content of register R2 with the rightmost bit forced to zero. The instruction immediately following the branch instruction is executed before the target instruction is executed.

The extended branch instructions based on **bnbrx** are:

brx	Branch Using Register with Execute
bgerx	Branch on Greater Than or Equal Using Register with Execute
bnerx	Branch on Not Equal Using Register with Execute
blerx	Branch on Less Than or Equal Using Register with Execute
bccrx	Branch on Carry Bit Clear Using Register with Execute
bvcrx	Branch on Overflow Clear Using Register with Execute
btcrx	Branch on Test Bit Clear Using Register with Execute.

Each of these instructions implicitly tests a CS bit, so that you do not have to specify the bit to test. For example, **brx** is equivalent to **bnbrx 0**. Figure 4-2 on page 4-13 explains the correspondence between **bbrx** and the extended branches.

For the extended branches, if the implicitly specified Condition Status bit is zero at execution, the updated instruction address is replaced by the content of register R1 with the rightmost bit forced to zero. The instruction immediately following the branch instruction is executed before the target instruction is executed.

Format:

bnbrx I1,R2 brx R1 bgerx R1 bnerx R1
blerx R1 bccrx R1 bvcrx R1 btcrx R1

for **bnbrx**

E9	IE	R2
0	8	12 15

for extended branches

E9	IE	R1
0	8	12 15

Remarks:

- If the implicitly or explicitly specified Condition Status bit is one, the updated instruction address is unaltered.
- For **bnbrx**, I1 must have an integer value of 0, 1, 2, 3, 4, 6, or 7.
- For **bnbrx**, IE = I1+8. For the extended branches, there is no I1 operand; the assembler inserts the proper IE value into the object code.
- Note that **brx** always branches, since the PZ bit is always zero.

Examples: *Example 1*

```
          # assume GPR 12 holds 0x2080 0100

                 clrcb 15,10
@1800 0000  here: bnbrx 2,12
@1800 0004       setb 6,13

          # first setb executes
          # updated IAR now 0x2080 0100
          # branch occurs to there

              .

              .
@2080 0100  there:
```

Example 2

```
          # assume GPR 12 holds 0x3080 0100

                clrcb 15,10
@3800 0000  pan: bnerx 12
@3800 0004      setb 6,13

          # first setb executes
          # updated IAR now 0x3080 0100
          # branch occurs to fire

              .

              .
@3080 0100  fire:
```

See Also: "Branch Instructions" on page 4-9

bnbx Branch on Not Condition Bit Immediate with Execute

bx, bccx, bgex, blex, bnex, btcx, bvcx Extended Branches also

Purpose: For the **bnbx** instruction, you explicitly specify a Condition Status bit to be tested. The assembler subtracts the address of the branch instruction from the address of A2. The assembler then divides the result by two to obtain BI, the displacement in halfwords. If the Condition Status bit specified by I1 is zero at execution, BI is sign extended and shifted left one bit to form the displacement in bytes. This byte displacement is added to the branch instruction address. The result then replaces the updated instruction address. The instruction immediately following the branch instruction is executed before the target instruction is executed.

The extended branch instructions based on **bnbx** are:

bx	Branch With Execute
bgex	Branch on Greater Than or Equal With Execute
bnex	Branch on Not Equal With Execute
blex	Branch on Less Than or Equal With Execute
bccx	Branch on Carry Bit Clear With Execute
bvcx	Branch on Overflow Clear With Execute
btcx	Branch on Test Bit Clear With Execute.

Each of these instructions implicitly tests a CS bit, so that you do not have to specify the bit to test. For example, **bx** is equivalent to **bnbx 0**. Figure 4-2 on page 4-13 explains the correspondence between **bbx** and the extended branches.

For the extended branches, the assembler subtracts the address of the branch instruction from the address of A1. The assembler then divides the result by two to obtain BI, the displacement in halfwords. If the implicitly specified Condition Status bit is zero at execution, BI is sign extended and shifted left one bit to form the displacement in bytes. This byte displacement is added to the branch instruction address. The result then replaces the updated instruction address. The instruction immediately following the branch instruction is executed before the target instruction is executed.

Format:

bnbx I1,A2 bx A1 bgex A1 bnex A1
blex A1 bccx A1 bvcx A1 btcx A1

89	IE	BI
0	8	12 31

Remarks:

- If the implicitly or explicitly specified Condition Status bit is one, the updated instruction address is unaltered.
- For **bnbx**, I1 must have an integer value of 0, 1, 2, 3, 4, 6, or 7.
- For **bnbx**, IE = I1+8. (See Figure 4-1 on page 4-12.) For the extended branches, there is no I1 operand; the assembler inserts the proper IE value into the object code.
- Note that **bx** always branches, since the PZ bit is always zero.

Examples: *Example 1*

```
                     clrcb 15,10
@1000 0100      here: bnbx 2,there

                # displacement between here and there is 0x200 halfwords
                # BI is 0x00200

@1000 0104           setb 6,13

                # at execution, this instruction executes
                # updated IAR replaced by 0x1000 0500
                # branch occurs to there

@1000 0500      there:
```

Example 2

```
                    clrcb 15,10
@3000 0100      pan: bnex fire

                # displacement between pan and fire is 0x200 halfwords
                # BI is 0x00200

@3000 0104          setb 6,13

                # at execution, this instruction executes
                # updated IAR replaced by 0x3000 0500
                # branch occurs to fire

                    .
                    .
@3000 0500      fire:
```

See Also: "Branch Instructions" on page 4-9

clrcb on page 4-79

Compare c

Purpose: The contents of registers R1 and R2 are compared. Condition Status bit LT is set if the content of register R1 is algebraically less than the content of register R2. Condition Status bit GT is set if the content of register R1 is algebraically greater than the content of register R2. Condition Status bit EQ is set if the content of register R1 equals the content of register R2.

Format: c R1,R2

B4	R1	R2
0	8	12 15

Remarks:

- Registers R1 and R2 are both treated as 32-bit signed algebraic quantities.
- Condition Status bits LT, EQ, and GT are affected.

Example:

```
        # assume GPR 4 has value 0xFFFF FFE7
        # (representing decimal -25)
        # and GPR 5 has value 0x0000 0011
        # (representing decimal 17)
c 4,5
        # this sets CS bit LT to one
        # and sets bits EQ and GT to zero
```

See Also: “Arithmetic Instructions” on page 4-16

cal — Compute Address Lower Half

Purpose: D2 is sign extended and added to the word specified by 0/(R2). The result replaces the content of register R1.

Format: cal R1,D2(R2)

C8	R1	R2	D2	
0	8	12	16	31

Remarks:

- The word specified by 0/(R2) can be an address.
- This instruction can be used to load a register with an immediate value from -32,768 to 32,767.

Examples:

```
     # assume GPR 5 contains 0x0000 0900
cal 4,0x8FF0(5)
     # GPR 4 now contains 0xFFFF 98F0


cal 3,150(0)
     # GPR 3 now contains 0x0000 0096
```

See Also: "Address Computation Instructions" on page 4-8

Compute Address Lower Half 16-Bit **cal16**

Purpose: This instruction adds D2 to the lower half of 0/(R2) and puts the result into the lower half of register R1. The instruction then replaces the upper half of register R1 with the upper half of 0/(R2).

Format: cal16 R1,D2(R2)

C2	R1	R2	D2	
0	8	12	16	31

Remarks:

- This instruction is provided to assist in simulation of 16-bit architectures.
- D2 must be an absolute expression.
- This instruction can be used to load a constant into a register. The constant must have a value from 0 to 65,535.

Examples:

```
      # assume GPR 4 contains 0x0101 A000
cal16 5,x1234(4)
      # GPR 5 now contains 0x0101 B234


cal16 3,0x8000(0)
      # GPR 3 now contains 0x0000 8000
```

See Also: "Address Computation Instructions" on page 4-8

cau on page 4-71

cas — Compute Address Short

Purpose: The word specified by (R2) + 0/(R3) replaces the content of register R1.

Format: cas R1,R2,R3

6	R1	R2	R3
0	4	8	12 15

Remarks:

- The word specified by (R2) + 0/(R3) can represent an address.
- If R3 is specified as zero, **cas R1,R2,0** has the same effect as **lr R1,R2**.
- Specifying the **nop** instruction causes the assembler to generate the no-operation instruction **cas 0,0,0**.

Examples:

```
     # assume GPR 5 contains 0x0007 0800
     # and GPR 6 contains 0x002F 03F0
cas 4,5,6
     # GPR 4 now contains 0x0036 0BF0


cas 4,5,0
     # R3 is zero, so the value
     # of the operand defaults to zero.
     # GPR 4 now contains 0x0007 0800
```

See Also: "Address Computation Instructions" on page 4-8

Compute Address Upper Half — cau

Purpose: D2 is extended on the right with 16 zeroes, then added to the content of 0/(R2). The resulting word replaces the content of register R1.

Format: cau R1,D2(R2)

D8	R1	R2	D2	
0	8	12	16	31

Remarks:

- The word resulting from **cau** can specify an address.
- D2 must be an absolute expression.
- A **cal16** followed by a **cau** can be used to load any 32-bit constant into a register.

Examples:

```
     # assume GPR 6 contains 0x0000 4000
cau 7,0x0011(6)
     # now GPR 7 contains 0x0011 4000
     # (this value represents an address)


cau 7,0x0011
     # R2 is not specified, so the value
     # of R2 defaults to zero.
     # GPR 7 now contains 0x0011 0000, which
     # represents an address



     # the next example shows how to load
     # 0xF183 81FE into a register

cal16 7,0x81FE(0)  # GPR 7 now contains 0x0000 81FE
cau   7,0xF183(7)  # GPR 7 now contains 0xF183 81FE
```

See Also: "Address Computation Instructions" on page 4-8

cal16 on page 4-69

Compute Address 16-Bit ca16

Purpose: The lower half of register R1 is added to the lower half of register R2. The result replaces the lower half of register R1. The upper half of register R2 replaces the upper half of register R1.

Format: ca16 R1,R2

F3	R1	R2
0	8	12 15

Remarks:

- The new content of register R1 can represent an address.
- This instruction is provided to assist in simulation of 16-bit architectures.
- If overflow occurs out of the lower half of register R1, no bits are set in the CS register.

Example:

```
     # assume GPR 12 contains 0x0066 4400
     # assume GPR 13 contains 0x0A00 0030
ca16 12,13
     # now GPR 12 contains 0x0A00 4430
```

See Also: "Address Computation Instructions" on page 4-8

Purpose: For the long form, the content of register R1 is compared to the sign-extended I2 field. The Condition Status LT bit is set if the content of register R1 is algebraically less than the sign-extended I2 field. The GT bit is set if the content of register R1 is greater than the sign-extended I2 field. The EQ bit is set if the content of register R1 equals the sign-extended I2 field.

For the short form, the content of register R1 is compared to the I2 field extended on the left with 28 zeroes. The LT bit is set if the content of register R1 is algebraically less than the field I2 extended on the left with 28 zeroes. The GT bit is set if the content of register R1 is greater than the field I2 extended on the left with 28 zeroes. The EQ bit is set if the content of register R1 equals the field I2 extended on the left with 28 zeroes.

Format: ci R1,I2

long form

D4	0	R1	I2	
0	8	12	16	31

short form

94	R1	I2	
0	8	12	15

Examples:

```
        # assume GPR 6 contains 0x0000 000F

    ci 6,0xE
        # GT bit is set, since 0x0000 000F > 0x0000 000E.
        # op code 94 is generated

    ci 6,0x88FF
       # sign-extended I2 field contains 0xFFFF 88FF
       # (this represents -0x0000 7701)
       # GT bit is set, since 0x0000 000F > -0x0000 7701
       # op code D4 is generated
```

See Also: "Arithmetic Instructions" on page 4-16

Compare Logical cl

Purpose: The contents of registers R1 and R2 are compared. Condition Status bit LT is set if the content of register R1 is logically less than the content of register R2. Condition Status bit GT is set if the content of register R1 is logically greater than the content of register R2. Condition Status bit EQ is set if the content of register R1 equals the content of register R2.

Format: cl R1,R2

B3	R1	R2
0	8	12 15

Remarks: The contents of registers R1 and R2 are both treated as 32-bit unsigned quantities.

Example:

```
        # assume GPR 4 contains 0xFFFF 0000
        # and GPR 5 contains 0x7FFF 0000
cl 4,5
        # the GT bit is set
```

See Also: "Logical Operation Instructions" on page 4-18

cli — Compare Logical Immediate

Purpose: The content of register R1 is compared to the sign-extended I2 field. The Condition Status LT bit is set if the content of register R1 is logically less than the sign-extended I2 field. The GT bit is set if the register R1 is logically greater than the sign-extended I2 field. The EQ bit is set if the content of register R1 equals the sign-extended I2 field.

Format: cli R1,I2

D3	0	R1	I2	
0	8	12	16	31

Remarks: Condition Status bits LT, EQ, and GT are set according to the relative unsigned magnitudes of register R1 and the sign-extended I2 field. Thus I2 can never represent a negative number.

Example:

```
    # assume GPR 4 contains 0x0000 C000
cli 4,0x7000  # the GT bit is set
cli 4,0x8000  # the LT bit is set
```

See Also: "Logical Operation Instructions" on page 4-18

Clear Bit clrb

Purpose: Bit I2 in register R1 is set to zero.

Format: clrb R1,I2

lower half

99	R1	IN	
0	8	12	15

upper half

98	R1	IN	
0	8	12	15

Remarks:

- I2 must evaluate to an integer between 0 and 31 inclusive.
- The assembler examines I2 and automatically chooses the correct form (upper or lower half) of the instruction. If I2 ≤ 15, op code 98 is generated. If I2 > 15, op code 99 is generated.
- Condition Status bits LT, EQ, and GT are affected. If the result is a negative value, LT is set to one; if it is zero, EQ is set to one; and if it is positive and not zero, GT is set to one.

Examples:

```
        # assume GPR 2 contains 0x0008 8000
     clrb 2,12
        # op code 98 is generated
        # (object code for this instruction is 0x982C)
        # bit 12 in GPR 2 is set to zero
        # GT bit is set to one
        # GPR 2 is now 0x0000 8000


        # assume GPR 3 contains 0x0000 8000
     clrb 3,16
        # op code 99 is generated
        # (object code for this instruction is 0x9930)
        # bit 16 in GPR 3 is set to zero
        # EQ bit is set to one
        # GPR 3 now contains 0x0000 0000
```

See Also: "Logical Operation Instructions" on page 4-18

Clear Bit in the System Control Register **clrcb**

Purpose: Bit I2 in the lower half of System Control Register SCR1 is set to zero.

In unprivileged state, SCR1 can have one of two values: 0xA for the Multiplier Quotient register (SCR 10), or 0xF for the Condition Status register (SCR 15). Other SCR1 values may only be used in privileged state.

If SCR1 indicates the Condition Status register, then I2 signifies the following:

I2= 0-7	Reserved
I2= 8	No operation (PZ bit is always zero)
I2= 9	Clear the Less Than condition bit
I2= A	Clear the Equal condition bit
I2= B	Clear the Greater Than condition bit
I2= C	Clear the Carry Zero condition bit
I2= D	Reserved
I2= E	Clear the Overflow condition bit
I2= F	Clear the Test bit

If SCR1 indicates the Multiplier Quotient register, then I2 may have any value from decimal 0 through 15 inclusive.

Format: clrcb SCR1,I2

95	SCR1	I2	
0	8	12	15

Remarks:

- If the selected bit of the SCR is a reserved bit, it is not set to a predictable value.
- If the specified SCR is the IAR (SCR 13), the results of **clrcb** are unpredictable.

Example:

```
clrcb 0xF,0xE
   # clears the Overflow bit in the CS register
```

See Also: "System Control Register Manipulation Instructions" on page 4-20

Hardware Technical Reference to learn how this instruction affects other SCRs

clz — Count Leading Zeroes

Purpose: The content of register R1 is replaced by the binary representation of the number of leading zeroes in the lower half of register R2 (that is, the number of binary zeroes to the left of the leftmost one bit in the lower half of register R2).

Format: clz R1,R2

F5	R1	R2
0	8	12 15

Remarks: If the lower half of register R2 is equal to zero, the content of register R1 is replaced by the binary representation of 16.

Example:

```
    # assume GPR 4 contains value 0x0001 0699
    # (there are five leading binary zeroes in
    # the lower half of this value)
clz 12,4
    # now GPR 12 contains 0x0000 0005
```

See Also: "Logical Operation Instructions" on page 4-18

Divide Step d

Purpose: The content of register R2 is added to or subtracted from (R1)//(bit 0 of MQ) depending on whether the signs of registers R1 and R2 disagree or agree.

The 32 rightmost bits of the result replace the content of register R1. The MQ is shifted left one position and bit 31 of the MQ is set to one if and only if the sign of the 33-bit result equals the sign of register R2. Condition Status bit C0 is set to one if the sign of the 33-bit result equals the sign of the content of register R2. Bit OV is set to one if the sign of the 33-bit result equals the sign of the content of register R1.

Format: d R1,R2

B6	R1	R2
0	8	12 15

Example: The Divide Step instruction may be used to construct algorithms for dividing one number by another. The following example describes an algorithm for dividing a 32-bit dividend by a 32-bit divisor. The operands are in two's complement representation.

PROBLEM

Divide X by Y giving quotient Q and remainder R where X, Y, Q and R are 32-bit numbers and Y is not equal to zero, +1, or -1.

INITIAL CONDITIONS

Set general-purpose register RB to the propagated sign of X (zero if X is non-negative, -1 if X is negative). This can be accomplished by loading RB with X and executing a **saril6 R1,15** instruction. Load Y into R2. Load X into MQ.

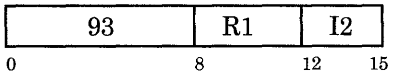
ALGORITHM

Issue the Divide Step instruction with operands R1 and R2 thirty-two times. If at this point the signs of R1 and R2 differ, do an **a R1,R2** (the sum will be put into R1). After this test and possible modification of R1, R1 contains the preliminary remainder. The MQ contains the 32 rightmost bits of the preliminary quotient. The final quotient and remainder are either equal to the preliminary quotient and remainder or are found by adding one to the preliminary quotient and subtracting the divisor, R2, from the preliminary remainder.

See Also: "Arithmetic Instructions" on page 4-16

Purpose: I2, extended on the left with 28 zeroes, is subtracted from the content of register R1. The result is placed into register R1.

Format: dec R1,I2



Examples:

```
  # assume GPR 4 contains 0x0000 100F
dec 4,0xF
  # now GPR 4 contains 0x0000 1000


  # assume GPR 5 contains 0x0000 000C
dec 5,0xF
  # now GPR 5 contains 0xFFFF FFFD, a
  # negative number
  # representing 0xC - 0xF
```

See Also: **inc** on page 4-84

"Address Computation Instructions" on page 4-8, "Arithmetic Instructions" on page 4-16

Extend Sign **exts**

Purpose: The content of the lower half of register R2 is sign extended. The result replaces the content of register R1.

Format: exts R1,R2

B1	R1	R2
0	8	12 15

Remarks: Condition Status bits LT, EQ and GT are affected.

Examples:

```
          # assume GPR 5 holds 0x0000 88A0
exts 4,5
          # now GPR 4 holds 0xFFFF 88A0
          # LT bit is set to one

          # assume GPR 7 holds 0xBBBB 7888
exts 6,7
          # now GPR 6 holds 0x0000 7888
          # GT bit is set to one
```

See Also: "Arithmetic Instructions" on page 4-16

Purpose: I2, extended on the left with 28 zeroes, is added to the content of register R1. The result is placed into register R1.

Format: inc R1,I2

91	R1	I2
0	8	12 15

Examples:

```
  # assume GPR 4 contains 0x0000 F006
inc 4,0x8
  # now GPR 4 contains 0x0000 F00E


  # assume GPR 5 contains 0xFFFF FFF4
inc 5,0xE
  # now GPR 5 contains 0x0000 0002
  # sum was >32 bits;
  # high-order overflow is lost
```

See Also: **dec** on page 4-82

"Address Computation Instructions" on page 4-8, "Arithmetic Instructions" on page 4-16

Warning: Using this instruction on the RT PC may cause conflicts with the AIX Operating System. If you use this instruction in unprivileged state, the machine will receive a hardware interrupt without precise indication of the address of the offending instruction.

Purpose: The content of register R1 is replaced by data transferred from the I/O device selected by the effective address 0/(R2) + 16 zeroes//D2. Bits 8 through 31 of the 32-bit effective address are interpreted as the I/O device address. Bits 0 through 7 of the effective address must be zero.

Format: ior R1,D2(R2)

CB	R1	R2	D2	
0	8	12	16	31

Remarks: This is not a privileged instruction. However, ordinary applications programs should never use **ior** in unprivileged state.

Example:

```
   # read a status register of a
   # device at address 0x88
ior 5,0x88(0)
   # GPR 5 now contains the value
   # read in from the status register
```

See Also: "Processor I/O Instructions" on page 4-20

Hardware Technical Reference

Warning: Using this instruction on the RT PC may cause conflicts with the AIX Operating System. If you use this instruction in unprivileged state, the machine will receive a hardware interrupt without precise indication of the address of the offending instruction.

Purpose: The content of register R1 is transferred to the I/O device selected by the effective address 0/(R2) + 16 zeroes//D2. Bits 8 through 31 of the 32-bit effective address are interpreted as the I/O device address. Bits 0 through 7 of the effective address must be zero.

Format: iow R1,D2(R2)

DB	R1	R2	D2
0	8	12	16 31

Remarks: This is not a privileged instruction. However, ordinary applications programs should never use **iow** in unprivileged state.

Example:

```
     # write zeroes to a device
     # at address 0x0020 0008
cau  2, 0x0020(0)
lis  1,0
iow  1,8(2)
```

See Also: "Processor I/O Instructions" on page 4-20

Hardware Technical Reference

Load l

Purpose: For the long form, the content of register R1 is replaced by the word in storage addressed by 0/(R2) plus the sign-extended D2 field. D2 is the number of bytes displacement.

For the short form, you specify D2 as the number of bytes displacement, but the assembler converts D2 to a word displacement and puts the word displacement into the object code. At runtime, the content of register R1 is replaced by the word in storage addressed by 0/(R2) + 26 zeroes//D2//2 zeroes.

Format: l R1,D2(R2)

long form

CD	R1	R2	D2
0	8	12	16 31

short form

7	D2	R1	R2
0	4	8	12 15

Remarks:

- The assembler examines D2 and chooses the correct form (short or long) of the instruction. The short form (op code 7) is generated if D2 ≤ 6 bits *and* D2′s last two bits are zero. The long form (op code CD) is generated if D2′s last two bits are not zero *or* 6 < D2 ≤ 16 bits.
- The effective address formed from D2 + 0/(R2) will have its low order two bits forced to zero.
- For the short form, D2 is always a positive displacement from the word addressed by (R2).

Examples:

Example 1

```
@0000 5044   .long OxCCCCDDDD
               # assume this is the word you want
                   .
                   .
               # assume GPR7 holds 0x0000 5000
             l 5,0x0044(7)
               # Puts OxCCCCDDDD into GPR 5.
               # This generates op code CD.
```

Example 2

```
@0000 5004   .long OxEEEEFFFF
               # assume this is the word you want
                   .
                   .
               # assume GPR 7 contains 0x0000 5000
             l 5,4(7)
               # Puts OxEEEEFFFF into GPR 5.
               # This generates op code 7.
```

Example 3

```
        .text
.main:        # this is generated
              # by the C compiler
          .
          .
        .using  _main,11
              # assume GPR 11 has been loaded
              # with the address of _main
          .
          .
        l  5,one
              # this generates the short form
              # object code in hex is 715B
        l  6,test
              # this generates the long form
              # object code in hex is CD6B 006C
          .
          .
        .data   3  # this is the constant pool
_main:  .long  .main
one:    .long   1
big:    .space  100
test:   .long   0
```

See Also: "Storage Access Instructions" on page 4-6

Purpose: For the long form, character C3 of register R1 is replaced by the character of storage addressed by 0/(R2) plus the sign-extended D2 field.

For the short form, character C3 of register R1 is replaced by the character of storage addressed by 0/(R2) + 28 zeroes//D2.

In either case, characters C0 through C2 (that is, the three upper bytes) of register R1 are set to zeroes.

Format: lc R1,D2(R2)

long form

CE	R1	R2	D2
0	8	12	16 31

short form

4	D2	R1	R2
0	4	8	12 15

Remarks: For long and short forms, you specify D2 as the number of bytes displacement. The assembler examines D2 and chooses the correct form (short or long) of the instruction. If 4 < D2 ≤ 16 bits, op code CE is generated. If D2 ≤ 4 bits, op code 4 is generated. In either case, the assembler inserts a byte displacement into the D2 field of the object code.

Examples:

Example 1

```
@0000 5044   .byte 0x55
      .
      .
               # assume GPR 7 holds 0x0000 5000

             lc 5,0x0044(7)

               # character at @0000 5044 is 0x55
               # GPR 5 now holds 0x0000 0055
               # op code CE is generated
```

Example 2

```
@0000 5002   .short 0x1122
      .
      .
               # assume GPR 7 holds 0x0000 5000

             lc 5,2(7)

               # character at @0000 5002 is 0x11
               # GPR 5 now holds 0x0000 0011
               # op code 4 is generated
```

Example 3

```
        .text
.main             # this is generated
                  # by C compiler
          .
          .
        .using _main,11
                  # assume GPR 11 has been loaded
                  # with the address of _main
          .
          .
        lc     5,one
                  # this generates the short form
                  # object code in hex is 445B
        lc     6,test
                  # this generates the long form
                  # object code in hex is CE6B 006C
          .
          .
        .data  3          # constant pool
_main:  .long  .main
one:    .byte  1
        .space 3
big:    .space 100
test:   .byte  0
```

See Also: "Storage Access Instructions" on page 4-6

Load Half lh

Purpose: If D2 is not 0, the lower half of register R1 is replaced by the halfword of storage addressed by 0/(R2) plus the sign-extended D2 field. The assembler chooses the long form.

If D2 is 0, the lower half of register R1 is replaced by the halfword of storage addressed by the content of register R2. The assembler chooses the short form.

In either case, the upper half of register R1 is set to zeroes.

Format: lh R1,D2(R2)

long form

DA	R1	R2	D2	
0	8	12	16	31

short form

EB	R1	R2	
0	8	12	15

Remarks: The effective address formed by D2 + 0/(R2) will have its low-order bit forced to zero.

Example:

```
@0000 1A22  .short 0xFFEE
                 # assume this is the halfword you want
                          .
                          .
                 # assume GPR 7 contains 0x0000 1200

            lh 5,0x0822(7)

                 # This puts 0xFFEE into the lower half of
                 # GPR 5.  The upper half of GPR 5 is
                 # filled with zeroes, and
                 # op code DA is generated.
```

See Also: **lha** on page 4-94

"Storage Access Instructions" on page 4-6

Purpose: For the long form, the lower half of register R1 is replaced by the halfword of storage addressed by 0/(R2) plus the sign-extended D2 field. D2 is the number of bytes displacement.

For the short form, you specify D2 as the number of bytes displacement. The assembler converts D2 to a number of halfwords displacement and puts the halfword displacement into the D2 field of the object code. At runtime, the lower half of register R1 is replaced by the halfword of storage addressed by 0/(R2) + 27 zeroes//D2//0.

In either case, the high-order bit of the addressed halfword is extended through the upper half of register R1.

Format: lha R1,D2(R2)

long form

CA	R1	R2	D2
0	8	12	16 31

short form

5	D2	R1	R2
0	4	8	12 15

Remarks:

- The assembler examines D2 and chooses the correct form (short or long) of the instruction. The short form (op code 5) is generated if D2 ≤ 5 bits *and* if D2′s low-order bit is zero. The long form (op code CA) is generated if 5 < D2 ≤ 16 bits *or* if D2′s low-order bit is not zero.
- The effective address formed from D2 + 0/(R2) will have its low-order bit forced to zero at runtime.

Examples:

Example 1

```
@0000 1A22 .short 0xFFEE
                # assume this is the halfword you want
                   .
                   .
                # assume GPR 7 contains 0x0000 1200

           lha 5,0x0822(7)

                # The lower half of GPR 5 is replaced by
                # 0xFFEE.  The high-order bit of this
                # halfword (in this case, 1) is extended
                # through the upper half of GPR 5.
                # So now GPR 5 contains 0xFFFF FFEE.
                # Op code CA is generated.
```

Example 2

```
@0000 1202 .short 0x3344
                # assume this is the halfword you want
                   .
                   .
                # assume GPR 7 contains 0x0000 1200

           lha 6,2(7)

                # GPR 6 now holds 0x0000 3344
                # op code is generated
```

Example 3

```
        .text
.main            # this is generated
                # by C compiler
           .
           .
        .using  _main,11
                   # assume GPR 11 has been loaded
                   # with the address of _main
           .
           .
        lha     5,one
                   # this generates the short form
                   # object code in hex is 525B
        lha     6,test
                   # this generates the long form
                   # object code in hex is CA6B 006C
           .
           .
        .data   3      # constant pool
_main:  .long  .main
one:    .short  1
        .space  2
big:    .space  100
test:   .short  0
```

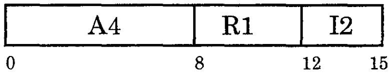
See Also: lh on page 4-93

"Storage Access Instructions" on page 4-6

Load Immediate Short lis

Purpose: The content of register R1 is replaced by the I2 field, extended on the left with 28 zeroes.

Format: lis R1,I2



Example:

```
lis 5,11
   #now GPR 5 holds 0x0000 000B
```

See Also: "Address Computation Instructions" on page 4-8

lm — Load Multiple

Purpose: The content of registers R1 through 15 are replaced, respectively, by the consecutive words in storage beginning at the address given by 0/(R2) plus the sign-extended D2 field.

Format: lm R1,D2(R2)

C9	R1	R2	D2	
0	8	12	16	31

Remarks: The effective address formed by D2 + 0/(R2) will have its low-order two bits forced to zero at runtime.

Example:

```
@0000 7700 .long 0x10002200,0x10003300,0x10004400
            .
            .
    # assume GPR 10 contains 0x0000 6000

         lm 13,0x1700(10)

    # GPR 13 now contains 0x1000 2200
    # GPR 14 now contains 0x1000 3300
    # GPR 15 now contains 0x1000 4400
```

See Also: "Storage Access Instructions" on page 4-6

Warning: User programs should not use this instruction in unprivileged state. Using **lps** outside of a virtual machine will generate a virtual interrupt program check with a privileged instruction exception.

Purpose: The content of the IAR is replaced by the word in main storage addressed by 0/(R2) plus the sign-extended D2 field. The Interrupt Control Status (ICS) is replaced by the content of the halfword in main storage at address 0/(R2) + the sign-extended D2 field + 4. The content of the Condition Status (CS) Register is replaced by the content of the halfword in main storage at address 0/(R2) + the sign-extended D2 field + 6.

I1 can have the value 0 or 1. If I1=0, interrupts may occur after the **lps** instruction executes. If I1=1, interrupts remain pending until the target of the **lps** instruction executes.

Format: lps I1,D2(R2)

D0	0	I1	R2	D2
0	8	11	12	16 31

Remarks:

- This is a privileged instruction; it must be executed in the processor's privileged state. In the hardware sense, this instruction can only be executed by the Virtual Resource Manager. However, the kernel can execute a virtual **lps**. See *Virtual Resource Manager Technical Reference* for information about the virtual **lps**.
- If the processor is on the Machine Check level when **lps** executes, the content of the MCS is set to zero. If the processor is on the Program Check level when **lps** executes, the content of the PCS is set to zero. See *Hardware Technical Reference* for details.

- In privileged state, you may use **lps** to return from an interrupt. You may also use **lps** to trace instruction execution by setting a bit in the IRB to generate an interrupt request before executing **lps**. The bit that is set should have an interrupt request priority greater than the processor priority that is loaded by **lps**. If the Interrupt Mask loaded by **lps** is zero, and if bit 11 of the **lps** instruction is one, an interrupt will occur after the **lps** target instruction executes.
- This instruction cannot be used as the subject of a branch with execute instruction. Doing so may put the processor in an unpredictable state.

See Also: *Virtual Resource Manager Technical Reference* for information on Interrupt Control Status (ICS) and virtual machines

Hardware Technical Reference for information about the PCS and MCS registers

Load Register lr

Purpose: The content of register R2 replaces the content of register R1.

Format: lr R1,R2

6	R1	R2	0

0 4 8 12 15

Remarks: An **lr** is the same as **cas R1,R2,0**.

Example:

```
   # assume GPR 5 holds 0x1100 5000
lr 4,5
   # now GPR 4 holds 0x1100 5000
```

See Also: "Address Computation Instructions" on page 4-8

Purpose: The incomplete product of the content of register R2 and bits 30 and 31 of the MQ register are formed in (R1)//MQ. A 34-bit sum is formed in accordance with the table below. The MQ is algebraically shifted right two positions with the two rightmost bits of the sum replacing bits 0 and 1 of the MQ. The content of register R1 is replaced by the 32 leftmost bits of the sum. Condition Status bit C0 is set to the complement of bit 30 of the MQ before the shift.

CS Bit C0	MQ Bit 30	MQ Bit 31	Algebraic Sum
0	0	0	(R1) + (R2)
0	0	1	(R1) + 2*(R2)
0	1	0	(R1) - (R2)
0	1	1	(R1) + 0
1	0	0	(R1) + 0
1	0	1	(R1) + (R2)
1	1	0	(R1) - 2*(R2)
1	1	1	(R1) - (R2)

Format: m R1,R2

```
|        E6        |    R1    |   R2   |
0                  8          12      15
```

Example: The Multiply Step instruction can be used to construct algorithms for multiplying two numbers. The following example describes an algorithm for multiplying a 32-bit multiplicand by a 16-bit multiplier. The operands are in two's complement representation.

EXAMPLE

Multiply X by Y giving Z, where X is a 32-bit number and Y is a 16-bit number.

INITIAL CONDITIONS

Load X into general purpose register R2. Load Y into the MQ. Set the content of general purpose register R1 to zero. Set Condition Status bit C0 to one. R1 and C0 can be initialized simultaneously by executing an **s R1,R1** instruction.

ALGORITHM

Issue the Multiply Step instruction with operands R1 and R2 eight times.

RESULT

The 16 rightmost bits of the product Z are in the MQ; the 32 leftmost bits are in register R1.

See Also: "Arithmetic Instructions" on page 4-16

mc03 Move Character Zero From Three

Purpose: Character C0 of register R1 is replaced by character C3 of register R2.

Format: mc03 R1,R2

F9	R1	R2
0	8	12 15

Example:

```
        # assume GPR 4 holds 0x1122 3344
        # assume GPR 5 holds 0x5566 7788
     mc03 4,5
        # GPR 4 now holds 0x8822 3344
```

See Also: "Move and Insert Instructions" on page 4-15

Move Character One From Three mc13

Purpose: Character C1 of register R1 is replaced by character C3 of register R2.

Format: mc13 R1,R2

FA	R1	R2
0	8	12 15

Example:

```
        # assume GPR 4 holds 0x1122 3344
        # assume GPR 5 holds 0x5566 7788
mc13 4,5
        # GPR 4 now holds 0x1188 3344
```

See Also: "Move and Insert Instructions" on page 4-15

mc23 — Move Character Two From Three

Purpose: Character C2 of register R1 is replaced by character C3 of register R2.

Format: mc23 R1,R2

FB	R1	R2
0	8	12 15

Example:

```
   # assume GPR 4 holds 0x1122 3344
   # assume GPR 5 holds 0x5566 7788
mc23 4,5
   # GPR 4 now holds 0x1122 8844
```

See Also: "Move and Insert Instructions" on page 4-15

Move Character Three From Zero mc30

Purpose: Character C3 of register R1 is replaced by character C0 of register R2.

Format: mc30 R1,R2

FD	R1	R2
0	8	12 15

Example:

```
   # assume GPR 4 holds 0x1122 3344
   # assume GPR 5 holds 0x5566 7788
mc30 4,5
   # GPR 4 now holds 0x1122 3355
```

See Also: "Move and Insert Instructions" on page 4-15

mc31 Move Character Three From One

Purpose: Character C3 of register R1 is replaced by character C1 of register R2.

Format: mc31 R1,R2

FE	R1	R2
0	8	12 15

Example:

```
        # assume GPR 4 holds 0x1122 3344
        # assume GPR 5 holds 0x5566 7788
    mc31 4,5
        # GPR 4 now holds 0x1122 3366
```

See Also: "Move and Insert Instructions" on page 4-15

Move Character Three From Two mc32

Purpose: Character C3 of register R1 is replaced by character C2 of register R2.

FF	R1	R2
0	8	12 15

Format: mc32 R1,R2

Example:

```
   # assume GPR 4 holds 0x1122 3344
   # assume GPR 5 holds 0x5566 7788
mc32 4,5
   # GPR 4 now holds 0x1122 3377
```

See Also: “Move and Insert Instructions” on page 4-15

mc33 Move Character Three From Three

Purpose: Character C3 of register R1 is replaced by character C3 of register R2.

Format: mc33 R1,R2

FC	R1	R2
0	8	12 15

Example:

```
      # assume GPR 4 holds 0x1122 3344
      # assume GPR 5 holds 0x5566 7788
mc33 4,5
      # GPR 4 now holds 0x1122 3388
```

See Also: "Move and Insert Instructions" on page 4-15

Move from System Control Register mfs

Purpose: The content of system control register SCR1 is placed in register R2.

In privileged state, SCR1 can have a value from 6 to 15 inclusive. In unprivileged state, SCR1 can have one of two values:

10—for Multiplier Quotient
15—for Condition Status Register.

Format: mfs SCR1,R2

96	SCR1	R2
0	8	12 15

Remarks:

- If the specified SCR has reserved bits, the corresponding bits of register R2 are set to unpredictable values.
- If the specified SCR is the IAR (SCR 13), the value loaded into register R2 is the address of the instruction immediately following the **mfs** instruction in main storage.
- See Chapter 1 for an explanation of other system control registers.

Example:

```
    # assume MQ register holds 0x0043 6211
mfs 0xA,9
    # now GPR 9 holds 0x0043 6211
```

See Also: "System Control Register Manipulation Instructions" on page 4-20

mftb — Move From Test Bit

Purpose: The bit of register R1 specified by the value of bits 27 through 31 of register R2 is set to the value of the Condition Status Test Bit.

Format: mftb R1,R2

BC	R1	R2
0	8	12 15

Example:

```
setcb 0xF,0xF  # sets test bit in CS to 1
   # assume GPR 4 holds 0xC100 666A
   # assume GPR 5 holds 0x1000 005D
mftb 4,5
   # bits 27-31 of GPR 5 have binary value 11101
   # so bit 29 of GPR 4 is set to 1
   # GPR 4 now holds 0xC100 666E
```

See Also: "Move and Insert Instructions" on page 4-15

Move From Test Bit Immediate mftbi

Purpose: Bit I2 of register R1 is set to the value of the Condition Status test bit.

Format: mftbi R1,I2

upper half

9C	R1	IN
0	8	12 15

lower half

9D	R1	IN
0	8	12 15

Remarks:

- The assembler examines I2 and automatically chooses the correct form (upper or lower half) of the instruction. If I2 ≤ 15, op code 9C is generated. If I2 > 15, op code 9D is generated.
- I2 must evaluate to an integer between decimal 0 and 31 inclusive.

Examples:

```
setcb 0xF,0xF
   # sets test bit in CS to 1
   # assume GPR 4 has 0xC100 666A
mftbi 4,29
   # op code 9D is generated
   # bit 29 of GPR 4 now set to 1
   # GPR 4 now holds 0xC100 666E



   # assume GPR 5 holds 0x0009 2222
mftbi 5,14
   # op code 9C is generated
   # bit 14 of GPR 5 now set to 1
   # GPR 5 now holds 0x000B 2222
```

See Also: "Move and Insert Instructions" on page 4-15

mts — Move To System Control Register

Purpose: The content of system control register SCR1 is replaced by the content of register R2.

In privileged state, SCR1 can have a value between 6 and 15 inclusive. In unprivileged state, SCR1 can have one of two values:

10—for Multiplier Quotient
15—for Condition Status Register.

Format: mts SCR1,R2

B5	SCR1	R2
0	8	12 15

Remarks:

- Any reserved bits in the specified SCR are not set to predictable values.
- If the specified SCR is the IAR (SCR 13), the results of the **mts** instruction are unpredictable.

Example:

```
   # assume GPR 11 holds 0x0700 3321
mts 15,0xB
   # If CS occupied entire register, the
   # register would hold 0x0700 3321.
   # But only last 8 bits of the CS
   # are defined (0x21), so bits
   # EQ and TB are set to one.
```

See Also: Chapter 1 for an explanation of other system control registers

"System Control Register Manipulation Instructions" on page 4-20

Move To Test Bit mttb

Purpose: The Condition Status Test Bit is set to the value of the bit of register R1 specified by the value of bits 27 through 31 of register R2.

Format: mttb R1,R2

BF	R1	R2
0	8	12 15

Example:

```
   # assume GPR 4 holds 0xC100 666A
   # assume GPR 5 holds 0x1000 005D
mttb 4,5
   # bits 27-31 of GPR 5 have binary 11101
   # bit 29 of GPR 4 is 0
   # so CS test bit is set to 0
```

See Also: "Move and Insert Instructions" on page 4-15

Purpose: The Condition Status Test Bit is set to the value of the bit in register R1 specified by I2.

Format: mttbi R1,I2

upper half

9E	R1	IN
0	8	12 15

lower half

9F	R1	IN
0	8	12 15

Remarks:

- The assembler examines I2 and automatically chooses the correct form (upper or lower half) of the instruction. If I2 ≤ 15, op code 9E is generated. If I2 > 15, op code 9F is generated.
- I2 must evaluate to an integer between 0 and 31 inclusive.

Examples:

```
  # assume GPR 4 holds 0xC100 666A
mttbi 4,29
  # op code 9F is generated
  # bit 29 of GPR 4 now set to 0
  # so CS test bit is set to 0


  # assume GPR 5 holds 0x0009 2222
mttbi 5,14
  # op code 9E is generated
  # bit 14 of GPR 5 now set to 0
  # so CS test bit is set to 0
```

See Also: "Move and Insert Instructions" on page 4-15

AND n

Purpose: The contents of registers R1 and R2 are ANDed. The result replaces the content of register R1.

Format: n R1,R2

E5	R1	R2
0	8	12 15

Remarks: Condition Status bits LT, EQ, and GT are affected.

Example:

```
          # assume GPR 4 has 0xFFF2 5730
          # assume GPR 5 has 0x7B41 92C0
       n 4,5
          # now GPR 4 has 0x7B40 1200
          # GT bit set to one
```

See Also: "Logical Operation Instructions" on page 4-18

nilo — AND Immediate Lower Half Extended Ones

Purpose: I3 is extended on the left with 16 ones, then ANDed with the content of register R2. The result replaces the content of register R1.

Format: nilo R1,R2,I3

C6	R1	R2	I3
0	8	12	16 31

Remarks: Condition Status bits LT, EQ, and GT are affected.

Example:

```
   # assume GPR 5 holds 0x7B41 92C0
nilo 4,5,0x5730
   # now GPR 4 holds 0x7B41 1200
   # GT bit set to one
```

See Also: "Logical Operation Instructions" on page 4-18

AND Immediate Lower Half Extended Zeroes nilz

Purpose: I3 is extended on the left with 16 zeroes, then ANDed with the content of register R2. The result replaces the content of register R1.

Format: nilz R1,R2,I3

C5	R1	R2	I3
0	8	12	16 31

Remarks: Condition Status bits LT, EQ, and GT are affected.

Example:

```
   # assume GPR 5 holds 0x7B41 92C0
nilz 4,5,0x5730
   # now GPR 4 holds 0x0000 1200
   # GT bit set to one
```

See Also: "Logical Operation Instructions" on page 4-18

Purpose: I3 is extended on the right with 16 ones, then ANDed with the content of register R2. The result replaces the content of register R1.

Format: niuo R1,R2,I3

D6	R1	R2	I3	
0	8	12	16	31

Remarks: Condition Status bits LT, EQ, and GT are affected.

Example:

```
   # assume GPR 5 holds 0x7B41 92C0
niuo 4,5,0x5730
   # now GPR 4 holds 0x5300 92C0
   # GT bit set to one
```

See Also: "Logical Operation Instructions" on page 4-18

AND Immediate Upper Half Extended Zeroes niuz

Purpose: I3 is extended on the right with 16 zeroes, then ANDed with the content of register R2. The result replaces the content of register R1.

Format: niuz R1,R2,I3

D5	R1	R2	I3	
0	8	12	16	31

Remarks: Condition Status bits LT, EQ, and GT are affected.

Example:

```
   # assume GPR 5 holds 0x7B41 92C0
niuz 4,5,0x5730
   # now GPR 4 holds 0x5300 0000
   # GT bit set to one
```

See Also: "Logical Operation Instructions" on page 4-18

nop — No Operation

Purpose: The IAR is incremented by two bytes, and the processor does nothing except proceed to the next instruction to be executed.

Format: nop

6	0	0	0
0	4	8	12 15

Remarks:

- Do not use **nop** to make software timing loops. Instead, you should rely on the system timer (**mts** and **mfs**) or real time clock facilities if you must write timing-dependent code.
- The **nop** instruction is the same as **cas 0,0,0**.
- The **nop** instruction does not affect the Condition Status bits.
- If the subject of a branch-with-execute instruction is only two bytes long, the assembler automatically inserts a **nop** after the subject instruction.

OR   o

Purpose: The contents of registers R1 and R2 are ORed. The result replaces the content of register R1.

Format: o R1,R2

E3	R1	R2
0	8	12 15

Remarks: Condition Status bits LT, EQ, and GT are affected.

Example:

```
   # assume GPR 4 holds 0xFFF2 5730
   # assume GPR 5 holds 0x7B41 92C0
o 4,5
   # now GPR 4 holds 0xFFF3 D7F0
   # LT bit set to one
```

See Also: "Logical Operation Instructions" on page 4-18

oil — OR Immediate Lower Half

Purpose: I3 is extended on the left with 16 zeroes, then ORed with the content of register R2. The result replaces the content of register R1.

Format: oil R1,R2,I3

C4	R1	R2	I3	
0	8	12	16	31

Remarks: Condition Status bits LT, EQ, and GT are affected.

Example:

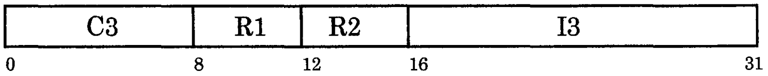
```
   # assume GPR 5 holds 0x7B41 92C0
oil 4,5,0x5730
   # now GPR 4 holds 0x7B41 D7F0
   # GT bit set to one
```

See Also: "Logical Operation Instructions" on page 4-18

OR Immediate Upper Half **oiu**

Purpose: I3 is extended on the right with 16 zeroes, then ORed with the content of register R2. The result replaces the content of register R1.

Format: oiu R1,R2,I3



Remarks: Condition Status bits LT, EQ, and GT are affected.

Example:

```
   # assume GPR 5 holds 0x7B41 92C0
oiu 4,5,0x5730
   # now GPR 4 holds 0x7F71 92C0
   # GT bit set to one
```

See Also: "Logical Operation Instructions" on page 4-18

onec — One′s Complement

Purpose: The content of register R1 is replaced by the one′s complement of the content of register R2.

Format: onec R1,R2

F4	R1	R2
0	8	12 15

Remarks: The LT, EQ, and GT Condition Status bits are affected.

Example:

```
   # assume GPR 5 holds 0x0036 8ACC
onec 4,5
   # now GPR 4 has oxFFC9 7533
   # LT bit set to one
```

See Also: "Arithmetic Instructions" on page 4-16

Subtract s

Purpose: The content of register R2 is subtracted from the content of register R1. The result is placed into register R1.

Format: s R1,R2

E2	R1	R2

0 8 12 15

Remarks: Condition Status bits LT, EQ, GT, C0, and OV are affected.

Example:

```
   # assume GPR 4 holds 0x006C 4930
   # assume GPR 5 holds 0x005A 3650
s 4,5
   # now GPR 4 holds 0x0012 12E0
   # GT and C0 bits set to one

   # assume GPR 4 holds 0x006C 4930
   # assume GPR 5 holds 0x005A 3650
s 5,4
   # now GPR 5 holds 0xFFED ED20
   # LT bit set to one
```

See Also: "Arithmetic Instructions" on page 4-16

sar — Shift Algebraic Right

Purpose: The content of register R1 is shifted right the number of bit positions specified by bits 26-31 of register R2. The vacated high-order positions are sign extended, that is, filled with bits equal to the original bit 0.

Format: sar R1,R2

B0	R1	R2
0	8	12 15

Remarks: Condition Status bits LT, EQ, and GT are affected.

Example:

```
   # assume GPR 4 holds 0x1234 5678
   # assume GPR 5 holds 0x00EE DBD4
sar 4,5
   # bits 26-31 of GPR 5 are binary 010100
   # which is decimal 20
   # GPR 4 now holds 0x0000 0123
   # GT bit set to one
```

See Also: "Shift Instructions" on page 4-19

Shift Algebraic Right Immediate — sari

Purpose: The content of register R1 is shifted right the number of bit positions specified by I2. The vacated high-order positions are sign extended, that is, filled with bits equal to the original bit 0.

Format: sari R1,I2

small form

A0	R1	IN

0 8 12 15

large form

A1	R1	IN

0 8 12 15

Remarks:

- I2 must evaluate to an integer between decimal 0 and 31 inclusive.
- The assembler examines I2 and generates the correct form (small or large) of the instruction. If I2 ≤ 15, op code A1 is generated. If I2 > 15, op code A0 is generated.
- Condition Status bits LT, EQ, and GT are affected.

Examples:

```
   # assume GPR 4 holds 0x1234 5678
sari 4,8
   # op code A0 is generated
   # now GPR 4 holds 0x0012 3456
   # GT bit set to one


   # assume GPR 5 holds 0x1234 5678
sari 5,20
   # op code A1 is generated
   # now GPR 5 holds 0x0000 0123
   # GT bit set to one
```

See Also: "Shift Instructions" on page 4-19

Purpose: The one's complement of the content of register R2 is added to the content of register R1. The value of Condition Status bit C0 is added to the result. The final result is placed in register R1.

Format: se R1,R2

F2	R1	R2
0	8	12 15

Remarks:

- This instruction allows multiple precision subtraction.
- Condition Status bits LT, EQ, GT, C0, and OV are affected.

Example:

```
   # assume GPR 4 holds 0x0044 6655
   # assume GPR 5 holds 0x0033 4422
setcb 0xC   # C0 now set to one
se 4,5
   # now GPR 4 holds 0x0011 2233
   # GT and C0 bits set to one
```

See Also: "Arithmetic Instructions" on page 4-16

Set Bit setb

Purpose: Bit I2 of register R1 is set to one.

Format: setb R1,I2

upper half

9A	R1	IN
0	8	12 15

lower half

9B	R1	IN
0	8	12 15

Remarks:

- The assembler examines I2 and generates the correct form (upper or lower half) of the instruction. If I2 ≤ 15, op code 9A is generated. If I2 > 15, op code 9B is generated.
- I2 must evaluate to an integer between decimal 0 and 31 inclusive.
- Condition Status bits LT, EQ, and GT are affected.

Examples:

```
   # assume GPR 4 holds 0x0037 0037
setb 4,12
   # op code 9A is generated
   # now GPR 4 holds 0x003F 0037
   # GT bit set to one


   # assume GPR 5 holds 0x0037 0037
setb 5,19
   # op code 9B is generated
   # now GPR 5 holds 0x0037 1037
   # GT bit set to one
```

See Also: “Logical Operation Instructions” on page 4-18

setcb — Set Bit in the System Control Register

Purpose: Bit I2 in the lower half of System Control Register SCR1 is set to one.

In privileged state, SCR1 can have a value from 6 to 15 inclusive. In unprivileged state, SCR1 can have one of two values: 10 for the Multiplier Quotient register, or 15 for the Condition Status.

If SCR1 is the Condition Status register, I2 has the following significance:

I2= 0-7	Reserved
I2= 8	No operation (PZ bit is always zero)
I2= 9	Set the Less Than condition bit
I2= A	Set the Equal condition bit
I2= B	Set the Greater Than condition bit
I2= C	Set the Carry Zero condition bit
I2= D	Reserved
I2= E	Set the Overflow condition bit
I2= F	Set the Test bit.

If SCR1 is the Multiplier Quotient register, I2 is any number from decimal 0 through 15 inclusive.

Format: setcb SCR1,I2

97	SCR1	I2
0	8	12 15

Remarks:

- If the selected bit of the SCR is a reserved bit, that bit is not set to a predictable value.
- If the specified SCR is the IAR (SCR 13), the results of **setcb** are unpredictable.

Example:

```
setcb F,F  # sets the Test bit in the CS register
```

See Also: "System Control Register Manipulation Instructions" on page 4-20

Hardware Technical Reference to learn how this instruction affects other SCRs

Subtract From sf

Purpose: The content of register R1 is subtracted from the content of register R2. The result is placed in register R1.

Format: sf R1,R2

B2	R1	R2
0	8	12 15

Remarks: Condition Status bits LT, EQ, GT, C0, and OV are affected.

Examples:

```
   # assume GPR 4 holds 0x006C 4930
   # assume GPR 5 holds 0x005A 3650
sf 5,4
   # now GPR 5 holds 0x0012 12E0
   # GT and C0 bits set to one

   # assume GPR 4 holds 0x006C 4930
   # assume GPR 5 holds 0x005A 3650
sf 4,5
   # now GPR 4 holds 0xFFED ED20
   # LT bit set to one
```

See Also: "Arithmetic Instructions" on page 4-16

Purpose: The content of register R2 is subtracted from the sign-extended I3. The result replaces the content of register R1.

Format: sfi R1,R2,I3

D2	R1	R2	I3	
0	8	12	16	31

Remarks: Condition Status bits LT, EQ, GT, C0, and OV are affected.

Examples:

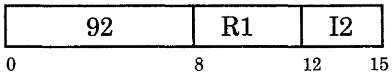
```
   # assume GPR 4 holds 0x006C 4930

sfi 5,4,0x8833
   # now GPR 5 holds 0xFF93 3F03
   # LT and C0 bits set to one

sfi 5,4,0x4930
   # now GPR 5 holds 0xFF94 0000
   # LT bit set to one
```

See Also: "Arithmetic Instructions" on page 4-16

Subtract Immediate Short sis

Purpose: I2 is extended on the left with 28 zeroes, then subtracted from the content of register R1. The result replaces the content of register R1.

Format: sis R1,I2



Remarks: Condition Status bits LT, EQ, GT, C0, and OV are affected.

Example:

```
   # assume GPR 4 holds 0x006C 4930
sis 4,8
   # now GPR 4 holds 006C 4928
   # GT and C0 bits set to one
```

See Also: "Arithmetic Instructions" on page 4-16

Purpose: The content of register R1 is shifted left the number of bit positions specified by bits 26 through 31 of register R2. Zeroes are supplied to the vacated low-order positions.

Format: sl R1,R2

BA	R1	R2
0	8	12 15

Remarks: Condition Status bits LT, EQ, and GT are affected.

Example:

```
   # assume GPR 4 holds 0x1234 5678
   # assume GPR 5 holds 0x00EE DBD4
sl 4,5
   # bits 26-31 of GPR 5 are binary 010100
   # which is decimal 20
   # now GPR 4 holds 0x6780 0000
   # GT bit set to one
```

See Also: "Shift Instructions" on page 4-19

Shift Left Immediate sli

Purpose: The content of register R1 is shifted left the number of bit positions specified by I2. Zeroes are supplied to the vacated low-order positions.

Format: sli R1,I2

small form

AA	R1	IN	
0	8	12	15

large form

AB	R1	IN	
0	8	12	15

Remarks:

- I2 is normally an integer between decimal 0 and 31 inclusive.
- The assembler examines I2 and chooses the correct form (small or large) of the instruction. If I2 ≤ 15, then IN = I2, and op code AA is generated. If I2 > 15, then IN = I2 - 16, and op code AB is generated.
- Condition Status bits LT, EQ, and GT are affected.

Examples:

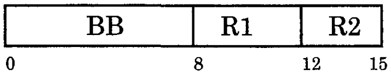
```
        # assume GPR 4 holds 0x1234 5678
     sli 4,20
        # op code AB is generated
        # GPR 4 now holds 0x6780 0000
        # GT bit set to one


        # assume GPR 5 holds 0x1234 5678
     sli 5,12
        # op code AA is generated
        # GPR 5 now holds 0x4567 8000
        # GT bit set to one
```

See Also: "Shift Instructions" on page 4-19

Shift Left Paired slp

Purpose: The content of register R1 is shifted left the number of bit positions specified by bits 26 through 31 of register R2. Zeroes are supplied to the vacated low order positions. The shifted content of register R1 is then placed in the pair of register R1.

Format: slp R1,R2



Remarks: Condition Status bits LT, EQ, and GT are affected.

Example:

```
   # assume GPR 4 holds 0x1234 5678
   # assume GPR 6 holds 0x00EE DBD4
slp 4,6
   # bits 26-32 of GPR 6 are binary 010100
   # which is decimal 20
   # now GPR 5 holds 0x6780 0000
   # GT bit set to one
```

See Also: "Shift Instructions" on page 4-19

slpi — Shift Left Paired Immediate

Purpose: The content of register R1 is shifted left I2 bit positions; zeroes are supplied to the vacated low order positions. The content of register R1 is then placed in the pair of register R1.

Format: slpi R1,I2

small form

AE	R1	IN
0	8	12 15

large form

AF	R1	IN
0	8	12 15

Remarks:

- I2 is normally an integer between decimal 0 and 31 inclusive.
- The assembler examines I2 and chooses the correct form (small or large) of the instruction. If I2 ≤ 15, then IN = I2, and op code AE is generated. If I2 > 15, then IN = I2 - 16, and op code AF is generated.
- Condition Status bits LT, EQ, and GT are affected.

Examples:

```
   # assume GPR 4 holds 0x1234 5678

slpi 4,20
   # op code AF is generated
   # now GPR 5 holds 0x6780 0000
   # GT bit set to one

slpi 4,12
   # op code AE is generated
   # now GPR 5 holds 0x4567 8000
   # GT bit set to one
```

See Also: "Shift Instructions" on page 4-19

Shift Right sr

Purpose: The content of register R1 is shifted right the number of bit positions specified by bits 26-31 of register R2. Zeroes are supplied to the vacated high-order positions.

Format: sr R1,R2

B8	R1	R2
0	8	12 15

Remarks: Condition Status bits LT, EQ, and GT are affected.

Example:

```
   # assume GPR 4 holds 0x1234 5678
   # assume GPR 5 holds 0x00EE DBD4
sr 4,5
   # bits 26-31 of GPR 5 are binary 010100
   # which is decimal 20
   # now GPR 4 holds 0x0000 0123
   # GT bit set to one
```

See Also: "Shift Instructions" on page 4-19

sri Shift Right Immediate

Purpose: The content of register R1 is shifted right the number of bit positions specified by I2. Zeroes are supplied to the vacated high-order positions.

Format: sri R1,I2

small form

A8	R1	IN
0	8	12 15

large form

A9	R1	IN
0	8	12 15

Remarks:

- I2 is normally an integer between decimal 0 and 31 inclusive.
- The assembler examines I2 and chooses the correct form (small or large) of the instruction. If I2 ≤ 15, then IN = I2, and op code A8 is generated. If I2 > 15, then IN = I2 - 16, and op code A9 is generated.
- Condition Status bits LT, EQ, and GT are affected.

Examples:

```
           # assume GPR 4 holds 0x1234 5678
        sri 4,20
           # op code A9 is generated
           # GPR 4 now holds 0x000 0123
           # GT bit set to one


           # assume GPR 5 holds 0x1234 5678
        sri 5,12
           # op code A8 is generated
           # GPR 5 now holds 0x0001 2345
           # GT bit set to one
```

See Also: “Shift Instructions” on page 4-19

Purpose: The content of the register R1 is shifted right the number of bit positions specified by bits 26-31 of register R2. Zeroes are supplied to the vacated high-order positions. The contents of register R1 are then placed in the pair of register R1.

Format: srp R1,R2

B9	R1	R2
0	8	12 15

Remarks: Condition Status bits LT, EQ, and GT are affected.

Example:

```
   # assume GPR 4 holds 0x1234 5678
   # assume GPR 6 holds 0x00EE DBD4
srp 4,6
   # bits 26-31 of GPR 6 are binary 010100
   # which is decimal 20
   # now GPR 5 holds 0x0000 0123
   # GT bit set to one
```

See Also: "Shift Instructions" on page 4-19

Shift Right Paired Immediate srpi

Purpose: The content of register R1 is shifted right the number of bit positions specified by I2, with zeroes supplied to the vacated high-order positions. The content of register R1 is then placed in the pair of register R1.

Format: srpi R1,I2

small form

AC	R1	IN
0	8	12 15

large form

AD	R1	IN
0	8	12 15

Remarks:

- I2 is normally an integer between decimal 0 and 31 inclusive.
- The assembler examines I2 and chooses the correct form (small or large) of the instruction. If I2 ≤ 15, then IN = I2, and op code AC is generated. If I2 > 15, then IN = I2 - 16, and op code AD is generated.
- Condition Status bits LT, EQ, and GT are affected.

Examples:

```
           # assume GPR 4 holds 0x1234 5678

        srpi 4,20
           # op code AD is generated
           # GPR 5 now holds 0x0000 0123
           # GT bit set to one

        srpi 4,12
           # op code AC is generated
           # GPR 5 now holds 0x0001 2345
           # GT bit set to one
```

See Also: "Shift Instructions" on page 4-19

Store st

Purpose: For the long form, 0/(R2) is added to the sign-extended D2 field. D2 is the number of bytes displacement, and the sum represents an address. The word in storage at this address is replaced by the content of register R1.

For the short form, you specify D2 is the number of bytes displacement, but the assembler converts D2 to a word displacement, and puts the word displacement into the object code. At runtime, 0/(R2) is added to 26 zeroes//D2//2 zeroes. The sum represents an address. The word in storage at this address is replaced by the content of register R1.

Format: st R1,D2(R2)

long form

DD	R1	R2	D2	
0	8	12	16	31

short form

3	D2	R1	R2	
0	4	8	12	15

Remarks:

- The assembler examines D2 and chooses the correct form (short or long) of the instruction. The short form (op code 3) is generated if D2 ≤ 6 bits *and* D2's last two bits are zero. The long form (op code DD) is generated if D2's last two bits are not zero *or* 6 < D2 ≤ 16 bits.
- The effective address formed from D2 + 0/(R2) will have its low order two bits forced to zero.

Examples:

Example 1

```
   # assume GPR 5 contains 0xAAAA BBBB
   # assume GPR 7 contains 0x0000 5000

st 5,0x0044(7)
   # The word at address 0x0000 5044 is
   # replaced by 0xAAAA BBBB.  This D2
   # value generates op code DD.

st 5,4(7)
   # The word at address 0x0000 5004 is
   # replaced by 0xAAAA BBBB.  This D2
   # value generates op code 3.
```

Example 2

```
        .text
.main:              # generated by C compiler
          .
          .
        .using  _main,11
                  # assume GPR 11 has been loaded
                  # with the address of _main
          .
          .
        st      5,one
                  # this generates the short form
                  # object code in hex is 315B
        st      6,test
                  # this generates the long form
                  # object code in hex is DD6B 006C
          .
          .
        .data   3       # constant pool
_main:  .long  .main
one:    .long   1
big:    .space  100
test:   .long   0
```

See Also: "Storage Access Instructions" on page 4-6

Store Character stc

Purpose: For the long form, 0/(R2) is added to the sign-extended D2 field. The sum represents an address. The character in storage at this address is replaced by character C3 of register R1.

For the short form, 0/(R2) is added to 28 zeroes//D2. The sum represents an address. The character in storage at this address is replaced by character C3 of register R1.

Format: stc R1,D2(R2)

long form

DE	R1	R2	D2
0	8	12	16 31

short form

1	D2	R1	R2
0	4	8	12 15

Remarks: D2 is the number of bytes displacement. The assembler examines D2 and chooses the correct form (short or long) of the instruction. If D2 ≤ 4 bits, then op code 1 is generated. If 4 bits < D2 ≤ 16 bits, then op code DE is generated.

Examples:

Example 1

```
   # assume GPR 5 holds 0x1234 5678
   # assume GPR 7 holds 0x0000 5000

stc 5,0x0044(7)
   # op code DE is generated
   # character of storage at address
   # 0x0000 5044 is replaced by 0x78

stc 5,2(7)
   # op code 1 is generated
   # character of storage at address
   # 0x0000 5002 is replaced by 0x78
```

Example 2

```
        .text
.main             # generated by C compiler
          .
          .
        .using  _main,11
                  # assume GPR 11 has been loaded
                  # with the address of _main
          .
          .
        stc     5,one
                  # this generates the short form
                  # object code in hex is 145B
        stc     6,test
                  # this generates the long form
                  # object code in hex is DE6B 006C
          .
          .
        .data   3      # constant pool
_main:  .long  .main
one:    .byte   1
        .space  3
big:    .space  100
test:   .byte   0
```

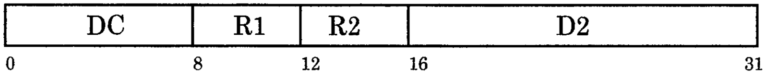
See Also: "Storage Access Instructions" on page 4-6

Purpose: For the long form, D2 is the number of bytes displacement. The halfword of storage addressed by 0/(R2) plus the sign-extended D2 field is replaced by the lower half of register R1.

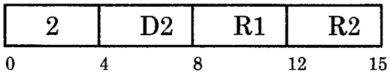
For the short form, you specify D2 as the number of bytes displacement. The assembler converts D2 to a number of halfwords displacement and puts the halfword displacement into the D2 field of the object code. At runtime, the lower half of register R1 is replaced by the halfword of storage addressed by 0/(R2) + 27 zeroes//D2//0.

Format: sth R1,D2(R2)

long form



short form



Remarks:

- The assembler examines D2 and chooses the correct form (short or long) of the instruction. The short form (op code 2) is generated if D2 ≤ 5 bits *and* if D2′s low-order bit is zero. The long form (op code DC) is generated if 5 < D2 ≤ 16 bits *or* if D2′s low-order bit is not zero.
- The effective address formed from D2 + 0/(R2) will have its low-order bit forced to zero at runtime.
- For the short form, D2 is the number of halfwords displacement at runtime. The assembler does not convert the halfword displacement back to a byte displacement.

Examples:

Example 1

```
   # assume GPR 5 contains 0xAAAA BBBB
   # and GPR 7 contains 0x0000 5000

sth 5,0x0044(7)
   # Halfword of storage addressed by 0x0000 5044
   # is replaced by xBBBB.  Op code DC is generated.

sth 5,2(7)
   # Halfword of storage addressed by 0x0000 5002
   # is replaced by xBBBB.  Op code 2
   # is generated.
```

Example 2

```
        .text
.main               # generated by C compiler
          .
          .
        .using  _main,11
                  # assume GPR 11 has been loaded
                  # with the address of _main
          .
          .
        sth     5,one
                  # this generates the short form
                  # object code in hex is 225B
        sth     6,test
                  # this generates the long form
                  # object code in hex is DC6B 006C
          .
          .
        .data  3      # constant pool
_main:  .long  .main
one:    .short 1
        .space 2
big:    .space 100
test:   .short 0
```

See Also: "Storage Access Instructions" on page 4-6

stm — Store Multiple

Purpose: The consecutive words in storage beginning at the address given by 0/(R2) plus the sign-extended D2 field are replaced, respectively, by the content of registers R1 through 15.

Format: stm R1,D2(R2)

D9	R1	R2	D2

0 8 12 16 31

Remarks: The effective address formed from D2 + 0/(R2) will have its low-order two bits forced to zero at runtime.

Example:

```
   # assume GPR 10 holds 0x0000 6000
   # assume GPR 13 holds 0x1000 2200
   # assume GPR 14 holds 0x1000 3300
   # assume GPR 15 holds 0x1000 4400

stm 13,0x1700(10)

   # three consecutive words in storage,
   # beginning at address 0x0000 7700,
   # are now 100022001000330010004400
   # (hex representation)
```

See Also: "Storage Access Instructions" on page 4-6

Warning: This instruction causes a trap to the operating system.

Purpose: The contents of the IAR replace the word that begins at hex address 190 in main storage (that is, real memory). The contents of the ICS replace the halfword that begins at hex address 194 in main storage. The contents of the CS replace the halfword that begins at hex address 196 in main storage. The low-order 16 bits of the 32-bit sum 0/(R1) + 16 zeroes//I1 are stored into the halfword that begins at hex address 19E in main storage.

The word beginning at hex address 198 in main storage replaces the contents of the IAR. The halfword beginning at hex address 19C in main storage replaces the contents of the ICS. Any reserved bits in the IAR and ICS are set to unpredictable values.

Format: svc I1(R1)

C0	0	R1	I1
0	8	12	16 31

Remarks:

- This instruction cannot be used as the subject of a branch with execute instruction. Doing so may put the processor in an unpredictable state.
- This instruction can be issued in unprivileged state; however, it is privileged in the sense that applications running in problem state do not ordinarily use it. An **svc** causes a trap to a real memory location where the Virtual Resource Manager resides, then causes the processor to go from unprivileged state to privileged state. The AIX Operating System kernel, if it exists, can request the Virtual Resource Manager to perform an **svc**. The I1 values necessary to do this are called "svc codes." However, applications programs should not use the **svc** to make a request to the kernel. Instead, applications programs should use AIX Operating System kernel system calls or subroutines.

See Also: "System Control Instructions" on page 4-21

Virtual Resource Manager Technical Reference for information on the **svc** codes

AIX Operating System Technical Reference for the system calls that application programs can use to issue requests to the kernel

tgte — Trap if Register Greater Than or Equal

Purpose: If the content of register R1 is logically greater than or equal to the content of register R2, the Trap bit of the Program Check Status register is set, and a program check occurs.

Format: tgte R1,R2

BD	R1	R2
0	8	12 15

Remarks: The AIX Operating System **sdb** command sets breakpoints by using **tgte**.

Example:

```
    # assume GPR 4 holds 0xF888 8888
    # assume GPR 5 holds 0x7999 9999
tgte 4,5
    # a trap occurs
```

See Also: "Trap Instructions" on page 4-15

Trap on Condition Immediate ti

Purpose: The content of register R2 and the value of the sign-extended I3 field are logically compared. If any of the trap conditions specified by bits 9 through 11 are met, the trap bit of the Program Check Status register is set, and a program check occurs.

Trap conditions are specified by I1, which occupies bits 8-11 as defined below. A trap is enabled if the bit is one, and disabled if the bit is zero.

Bit 8 Must always be zero.

Bit 9 Trap if register R2 is less than the value of the sign-extended I3 field.

Bit 10 Trap if register R2 is equal to the value of the sign-extended I3 field.

Bit 11 Trap if register R2 is greater than the value of the sign-extended I3 field.

Format: ti I1,R2,I3

CC	I1	R2	I3
0	8	12	16 31

Remarks:

- This is not a typical D format instruction, because it has an I1 field instead of an R1 field.
- When the comparisons are performed, the operands are treated as 32-bit unsigned integers.

Examples:

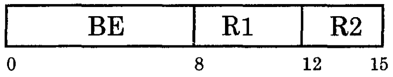
```
           # assume GPR 4 holds 0xEFFF 8555

        ti 2,4,0x8555
           # decimal 2 means bit 10 in I1 is one
           # no trap occurs

        ti 4,4,0x8555
           # decimal 4 means bit 9 in I1 is one
           # a trap occurs
```

See Also: "Trap Instructions" on page 4-15

Trap if Register Less Than tlt

Purpose: If the content of register R1 is less than the content of register R2, the Trap bit of the Program Check Status is set, and a program check occurs.

Format: tlt R1,R2



Example:

```
   # assume GPR 4 holds 0xEFFF 8555
   # assume GPR 5 holds 0xFFFF 8555
tlt 4,5
   # a trap occurs
```

See Also: "Trap Instructions" on page 4-15

tsh — Test and Set Half

Purpose: The lower half of register R1 is replaced by the halfword of storage addressed by 0/(R2) plus the sign-extended D2 field. The upper half of register R1 is set to zeroes. Immediately following the operation that reads the halfword of storage, the high-order byte of the selected halfword in storage is filled with binary ones. The low-order byte of the selected halfword is not changed.

Format: tsh R1,D2(R2)

CF	R1	R2	D2	
0	8	12	16	31

Remarks: The effective address formed by D2 + 0/(R2) will have its low-order bit forced to zero before accessing memory.

Example:

```
   # assume halfword 0xABCD begins
   # at address 0x0000 8800
   # assume GPR 4 holds 0x1122 3344
   # assume GPR 5 holds 0x0000 1100

tsh 4,0x7700(5)

   # now GPR 4 holds 0x0000 ABCD
   # now halfword 0xFFCD begins
   # at address 0x0000 8800
```

See Also: “Storage Access Instructions” on page 4-6

Two's Complement twoc

Purpose: The two's complement of the content of register R2 replaces the content of register R1.

Format: twoc R1,R2

E4	R1	R2
0	8	12 15

Remarks: Condition Status bits LT, EQ, GT, C0, and OV are affected.

Example:

```
   # assume GPR 5 holds 0x8765 4321
twoc 4,5
   # now GPR 4 holds 0x789A BCDF
   # GT bit is set to one
```

See Also: "Arithmetic Instructions" on page 4-16

Warning: Do not use this instruction in unprivileged state. If you use **wait** in unprivileged state, the machine generates a program check interrupt with a privileged instruction exception.

Purpose: This instruction places the processor in a wait state.

Format: wait 0,0

F0	0	0
0	8	12 15

Remarks:

- If you use this instruction in unprivileged state, an exception will be presented to the operating system.
- When the processor is in the wait state, it does not execute any instructions or make any storage accesses.
- The processor is removed from the wait state through the occurence of an interrupt, error, or power-on reset.

See Also: "System Control Instructions" on page 4-21

Exclusive OR x

Purpose: Registers R1 and R2 are exclusive ORed. The result replaces the content of register R1.

Format: x R1,R2

E7	R1	R2
0	8	12 15

Remarks: Condition Status bits LT, EQ, and GT are affected.

Example:

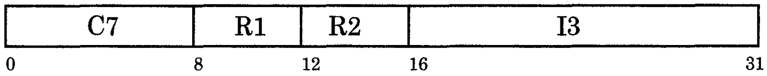
```
   # assume GPR 4 holds 0xFFF2 5730
   # assume GPR 5 holds 0x7B41 92C0
x 4,5
   # now GPR 4 holds 0x84B3 c5F0
   # LT bit is set to one
```

See Also: “Logical Operation Instructions” on page 4-18

xil Exclusive OR Immediate Lower Half

Purpose: I3 is extended on the left with 16 zeroes, then exclusive ORed with the content of register R2. The result replaces the content of register R1.

Format: xil R1,R2,I3



Remarks: Condition Status bits LT, EQ, and GT are affected.

Example:

```
   # assume GPR 5 holds 0x7B41 92C0
xil 4,5,0x5730
   # now GPR 4 holds 0x7B41 C5F0
   # GT bit is set to one
```

See Also: “Logical Operation Instructions” on page 4-18

Exclusive OR Immediate Upper Half — xiu

Purpose: I3 is extended on the right with 16 zeroes, then exclusive ORed with the content of register R2. The result replaces the content of register R1.

Format: xiu R1,R2,I3

D7	R1	R2	I3
0	8	12	16 ... 31

Remarks: Condition Status bits LT, EQ, and GT are affected.

Example:

```
   # assume GPR 5 holds 0x7B41 92C0
xiu 4,5,0x5730
   # now GPR 4 holds 0x2C71 92C0
   # GT bit is set to one
```

See Also: "Logical Operation Instructions" on page 4-18

Chapter 5. Pseudo-Ops

CONTENTS

About This Chapter

This chapter defines pseudo-operations and lists them by category. “Notational Conventions” on page 5-9 shows the notation used to describe pseudo-ops. A directory then lists pseudo-ops by mnemonic.

If you know the mnemonic of the pseudo-op, just look it up in the directory. If you don′t know the name or mnemonic of the pseudo-op, but you know what the pseudo-op should do, look at “Categories of Pseudo-Ops” on page 5-5.

What Is a Pseudo-Op?

A pseudo-operation, commonly called a pseudo-op, is an instruction to the assembler that does not generate any machine code. (The only exceptions are the **call** and **callr** pseudo-ops.) The assembler resolves pseudo-ops during assembly—unlike machine instructions, which are resolved only at runtime. Pseudo-ops are sometimes called assembler instructions, assembler operators, or assembler directives.

In general, pseudo-ops give the assembler information about data alignment, block and segment definition, and base register assignment. RT PC also supports pseudo-ops that give the assembler information about floating point constants, calling conventions, and the symbolic debugger (**sdb**).

The following pseudo-ops can change the contents of the assembler's location counter:

.align
.byte
call
callr
.comm
.double
.float
.lcomm
.long
.org
.short
.space

All the other pseudo-ops do not affect the value of the location counter.

Categories of Pseudo-Ops

Pseudo-ops can be grouped into the following functional categories.

Data Alignment

.align
.byte
.long
.short
.double
.float

These pseudo-ops are typically used in the data section of a program. They may be used to create data areas to be used by a program, as shown by the example below.

```
greeting:  .long 'H,'O,'W,'D,'Y
           .
           .
           lm 11,greeting
```

The **.double** and **.float** pseudo-ops represent floating point constants. Both of these pseudo-ops have a **fcon** operand, which is described at "Notational Conventions" on page 5-9. Note that the C compiler builds floating point constants with **.byte**; the C compiler does not generate **.double** or **.float**.

Data Definition

.set
.double
.float
.byte
.long
.short

See page 5-28 and "Defining a Symbol with a Pseudo-op" on page 2-14 for a discussion of **.set**.

Storage Definition

.space

Addressing within a Source Module (Base Registers)

.drop
.using

Direct Addressing

.direct

Assembler Section Definition

.comm
.data
.lcomm
.text

These pseudo-ops define assembler language sections, which ultimately define runtime segments.

External Symbol Definition

.globl

Symbol Table Entries for sdb

.bb
.bf
.eb
.ef
.eos
.file
.function
.line
.stab
.staba
.stabs
.stabt

The C compiler generates these pseudo-ops when you request the **cc** command with a **-g** flag and a C language input file. These pseudo-ops give symbol table information to the symbolic debugger, and have no other effect on assembly. IBM documentation does not discuss the **sdb** pseudo-ops, since they should only be inserted by a compiler, not by programmers.

Note: If you want to insert these **sdb** pseudo-ops into assembler language source code that does not already contain them, you should use your assembler source code as a C language function. Then, write a separate C file with a **main** calling the function that is written in assembler source.

For information on using **sdb**, see *AIX Operating System Programming Tools and Interfaces.*

Optimization Information

.copt

The C compiler generates this pseudo-op when you request the **cc** command with the **-O** flag and a C language input file. This pseudo-op passes information to the C optimizer, and has no effect on assembly. IBM documentation does not discuss the **.copt** pseudo-op, since it should only be inserted by a compiler, not by programmers.

Support for Calling Conventions

call
callr

The assembler expands these pseudo-ops into a series of instructions and pseudo-ops that call subroutines. Unlike other pseudo-ops, **call** and **callr** do not begin with a period, and they do generate machine code.

See "Subroutine Linkage and System Calls" on page 6-10 for the context in which these pseudo-ops are used.

Notational Conventions

All spaces are required unless otherwise specified. A space may optionally occur after a comma. Spaces are not allowed before a comma.

Some examples of pseudo-ops may not show labels. However, you can put a label in front of a pseudo-op statement just as you would for a machine instruction statement.

The following notational conventions are used to describe pseudo-ops:

[ ] (brackets)
Enclose optional operands.

name Any valid label.

R A general purpose register. R is an expression that evaluates to an integer between 0 and 15 inclusive.

N An expression that evaluates to an integer.

exp Unless otherwise noted, *exp* signifies a relocatable, constant, or absolute expression.

fcon A floating point constant. In 032 Microprocessor assembler language, an **fcon** consists of four parts in order:

integer part	Must be one or more digits.
decimal point	Is optional.
fraction part	Must be one or more digits.
exponent part	Is optional. Consists of an **e** or **E**, followed by a + or -, followed by one or more digits.

You may omit either the integer part or the fraction part, but not both. For example, the following are valid **fcon** operands:

```
.45
1e+5
4E-11
.99E6
357.22e12
```

There is no bounds checking for the operand; the range of the operand is the range of the floating point hardware. Also note that a **fcon** for a **.double** can be bigger than an **fcon** for a **.float**. For example, **5e300** would be valid for **.double** but not for **.float**.

... (ellipsis)

When shown as the last operand, a series of the operands preceding the ellipsis. For example, **exp,exp, ...** denotes two or more expressions separated by commas.

When shown between two operands, an ellipsis signifies operands to be added by the programmer. For example, **expl,exp2, ... exp5** means that the programmer supplies expl, exp2, exp3, exp4, and exp5.

Directory of Pseudo-Ops

The pseudo-ops are listed in alphabetic order. If you do not know the name of the pseudo-op you need, refer to "Categories of Pseudo-Ops" on page 5-5.

The directory entry for each pseudo-op includes a purpose, remarks, and example. Some directory entries also have remarks about the pseudo-op.

.align

Purpose: If necessary, advance the current location counter until the low-order **N** bits are filled with the value 0. **N** is an expression that evaluates to the integer value of 0, 1, or 2.

Format: .align N

Remarks:

- If **N** evaluates to 0, alignment occurs on a byte boundary. If **N** evaluates to 1, alignment occurs on a halfword boundary. If **N** evaluates to 2, alignment occurs on a fullword boundary.
- The linker will not allow alignment on anything bigger than a fullword boundary (such as a page boundary) for modules that are linked together.
- This pseudo-op is normally used in the data section of an assembler language program.
- If **.align** is used in the text section, alignment occurs by padding with **nop** instructions.

Example:

```
       .data
       .byte  1
         # location counter now at odd number
       .align 1
         # location counter now at
         # next halfword boundary
       .byte  3,2
          .
          .
          .
       .text
          .
          .
       .align 2   # insure the bali and
                  # .long are aligned
                  # on a fullword boundary
       bali   1,cont
       .long  5004381
cont:  l  2,0(1)    # load the big constant
```

.byte

Purpose: Assemble the values represented by the **exp** expressions into consecutive bytes.

Format: .byte exp,exp, ...

Remarks:

- The **exps** cannot contain externally defined symbols.
- If an **exp** is longer than one byte, it will be truncated.

Example:

```
                 .set olddata,0xCC
                      .
                      .
@2000 0000  mine: .byte 0x3F,0x7+0xA,olddata,0xFF
                      # load GPR 1 with 0x20000000
                 l 2,0(1)
                      # GPR 2 now holds 0x3F11CCFF
```

call

Purpose: Calls a subroutine. The label is the name of the subroutine being called. If the label is a C language subroutine, the label includes the leading period (.).

The pcp_address is the hex address of the pointer to the called routine's constant pool. This operand can be expressed as a label, or as a base and displacement of the form **D2(R2)**. However, any labels specified must be covered by a **.using**.

The number_words is the number of words required to store all parameters passed between the calling and the called routine. This value is used only by debuggers such as **sdb**. The debugger uses this value to display procedure parameters when showing information about the call. If debugger information is not being collected, this value is zero.

Format: call label, pcp_address, number_words

Remarks:

- The assembler expands the **call** pseudo-op into the following series of statements:

```
balix  15, .label          # call the routine
l      0, pcp_address     # get the routine's constant pool pointer
.byte  0x08, number_words # number of words of parameters passed --
                           # .byte acts as a no-op with operands
```

The **.byte** statement is generated only when the **number_words** operand was not zero (that is, only when **sdb** information was being gathered). If **number_words** was zero, then the **.byte** statement is not generated.

If the target of the **balix** is not within a megabyte of the call, then the branch target cannot be resolved. This could happen if the load module's text segment is larger than a megabyte, or if the target is in a shared library that is mapped into some other segment. In either case, the linkage editor changes **balix** into **balax** to a special sequence of code at a fixed location in the RT PC kernel. With the following code, this linkage routine uses the contents of the first word of the called routine's constant pool for the address of the distant entry point:

```
mts  10,14     # save register 14 into MQ register
lr   14,0      # make register 0 addressable
l    14,0(14)  # get destination address
brx   14       # branch to it and --
mfs  10,14     # -- restore register 14 from MQ register
```

- The linkage sequence assumes that register 0 points to the constant pool of the called routine, and that the first word of the constant pool is the address of the routine's entry point.

Example: The assembler expands this:

```
call  .foo, 12(14),3
```

into this:

```
balix  15,.foo
l      0,12(14)
.byte  0x08,3
```

See Also: "Subroutine Linkage and System Calls" on page 6-10

callr

Purpose: Calls a subroutine using a register that holds a pointer to the called routine's constant pool.

R is the register containing the address of the called routine's constant pool. R must not be 0. Number_words is the number of words required to store all parameters passed between the caller and the called routine. This value is used only by debuggers such as **sdb**. The debugger uses this value to display procedure parameters when showing information about the call. If debugger information is not being collected, this value is zero.

Format: callr R, number_words

Remarks:

- The assembler expands the **callr** pseudo-op into the following series of statements:

  ```
  l       15,0(R)             # get the routine's address
  balrx   15,15               # call the routine
  lr      0,R                 # put the routine's pcp pointer
                              # in register 0
  .byte   0x01, number_words # number of words of parameters passed --
                              # .byte acts as a no-op with operands
  ```

  The **.byte** statement is generated only when the **number_words** operand was not zero (that is, only when **sdb** information was being gathered). If **number_words** was zero, then the **.byte** statement is not generated.

- The linkage sequence assumes that register 0 points to the constant pool of the called routine, and that the first word of the constant pool is the address of the routine's entry point.

Example: The assembler expands this:

```
callr  8,3
```

into this:

```
l      15,0(8)
balrx  15,15
lr     0,8
.byte  0x08,3
```

See Also: "Subroutine Linkage and System Calls" on page 6-10

.comm

Purpose: Define a block of storage that will be common to more than one module. The block is named **name** and has a length of **exp** bytes.

Format: .comm name,exp

Remarks:

- The **exp** operand must be an absolute expression; **name** is relocatable.
- Use **.comm** when you know the size of a block of data that will be shared by two or more files, but you don't know whether that data will become initialized.
- The linker defines a common block of storage at link time. That is, the space declared with a **.comm** disappears at link time. If the data in the **.comm** space becomes initialized, it goes to the data runtime segment. If the **.comm** data is not initialized, it goes to the bss section. At load time, the bss section is created at the end of the data segment.
- If the original module or any linked modules contain more than one **.comm** definition of the same **name**, the assembler reserves space specified by the largest **exp**. The assembler does not generate an error message.
- By default, the linker defines common blocks in the bss section of the linked program. If you link in a module that defines **name** in the text or data assembler section, that module's definition of **name** will take precedence. The common block will then be defined in the text or data assembler section.

Example:

```
.comm  proc,5120
    # if proc is not defined elsewhere, proc
    # refers to 5120 bytes of storage in
    # the bss segment of the linked program
```

See Also: .data, .globl, .lcomm, .text

Chapter 3

Purpose: Switch to location counter **N** of the assembler language data section.

N is an expression that evaluates to an integer from 0 to 3 inclusive.

Format: .data [N]

Remarks:

- The default **N** value is 0.
- The data section holds initialized variables (data that gets changed when the program runs).
- When the assembler first encounters **.data N**, data location counter **N** initially receives a value of 0. The assembler increments this value in bytes as instructions are read. If **.data N** occurs again with the same **N** value, that data location counter **N** keeps the same value it had the last time it was used.

Examples:

```
.data # switches to location counter 0
      # of the data section

.data (sym+1)
      # valid values of sym are integers
      # between -1 and +2 inclusive
```

See Also: .comm, .lcomm, .text

Chapter 3

.direct

Purpose: The instructions in the same file as the **.direct** refer to an absolute address in memory. This pseudo-op allows direct addressing in instructions, as opposed to effective addressing.

Format: .direct

Remarks:

- The **.direct** can be anywhere in the file, and applies to all the instructions in the file.
- The assembler assumes that the program's text section will be linked in the lowest 32K of address space, so that 16-bit direct addresses can be used for text segment references not covered by **.using** statements.
- If any labels are subject to a **.direct** but not covered by a **.using**, the assembler assumes that the labels are based off location 0.
- The **.direct** pseudo-op is intended for use by kernel programmers, or for code that ultimately resides in virtual addresses 0 through 32767. User-made programs are always linked at hex locations 1000 0000 or higher.

Example:

```
.direct        # undefined symbols reside below 0x0000 7fff
bala  kern_call
```

See Also: .using

Chapter 3

Purpose: Representation of floating point constants.

Format: .double fcon

Remarks: Fullword alignment occurs as necessary.

Examples:

```
.double  3.4
.double  -77
.double  134E-12
.double  5e300
```

See Also: .float

"Data Definition" on page 5-5

.drop

Purpose: Stop using register **N** as a base register.

N is an expression that evaluates to an integer from 0 to 15 inclusive.

Format: .drop N

Remarks:

- You do not have to use the **.drop** pseudo-op before changing the base address with the **.using** pseudo-op.
- You do not have to use a **.drop** pseudo-op at the end of a program.

Example:

```
.using _subrA,1
    # r1 can now be used for addressing
    # with displacements calculated
    # relative to _subrA
  .
  .
  .
.drop 1
    # stop using r1
.using _subrB,1
    # now assembler calculates displacements
    # relative to _subrB
```

See Also: .using

Chapter 3

Purpose: Representation of a floating point constant.

Format: .float fcon

Remarks: Fullword alignment occurs as necessary.

Examples:

```
.float   3.4
.float   -77
.float   134E-12
```

See Also: .double

"Data Definition" on page 5-5

.globl

Purpose: Make **name** globally visible to the linker. **Name** is any label or symbol; it must be defined somewhere in the source file. **Name** will become available to any file that is linked to the file in which **.globl** occurs.

Format: .globl name

Remarks:

- If you don't use **.globl** for a symbol, then that symbol is, by default, only visible within the current assembly, and not to other modules that may later be linked. Also, for common blocks without a **.globl** declaration, the loader allocates memory in the bss section.
- If **name** is defined in the current assembly, its type and value arise from that definition, not the **.globl** definition.
- The **ld** command maps all common segments with the same name into the same memory. If in one of the segments the name is declared **.globl** and defined, this has the same effect as declaring the common symbols to be **.globl** in all segments. In this way, common memory can be initialized.

Example:

```
.globl main
main:
```

See Also: .comm

Purpose: **Name** is a label at the address specified by the current location counter for the bss assembler section. The contents of the bss location counter are incremented by **exp**.

This defines a local common block of storage named **name** with length **exp** bytes. At runtime, this storage block will be reserved when the bss section is allocated at the end of the data segment. This storage block is for uninitialized external data.

Format: .lcomm name,exp

Remarks:

- The **exp** operand must be an absolute expression that is defined in the first pass of the assembler; **name** is relocatable.
- To make **name** appear in the symbol table, you should declare **name** as a global symbol.
- The **.lcomm** and **.comm** pseudo-ops work together to define storage in a bss section.

Example:

```
.lcomm buffer,5120
    # can refer to this 5K of storage as "buffer"
```

See Also: .comm

.long

Purpose: Assemble expressions **exp** into consecutive fullwords.

Format: .long exp,exp, ...

Remarks:

- Fullword alignment occurs as necessary.
- The number of expressions is limited only by the amount of storage you have.

Example:

```
.long 24,3,fooble-333,0
```

Purpose: Set the value of the current location counter to **exp**. The **exp** is normally a relocatable expression.

Format: .org exp

Remarks:

- You cannot use **.org** to decrement a location counter.
- The assembler allows **exp** to be absolute, thereby producing non-relocatable code.

  Warning: If you use an absolute expression as an **.org** operand, make sure that you do not need the data you will overwrite. It is recommended that you use absolute **.org**s with extreme caution.

  If you use an absolute **.org**, do not run **cc** on the file containing the absolute **.org**. Instead, you should run **as** and **ld** separately on the file. Furthermore, when you run **ld** on the assembler output, you must do the following:

  - Specify the assembler output file as the first object file. That is, the assembler output file must be specified before the **crt0.o** file.
  - Specify the **-T** flag with an address equal to the address used in the absolute **.org**.

  Instead of using an absolute **.org**, you may be able to achieve the desired effect by specifying an absolute origin on the **ld** command line. See “Linking with ld” on page 6-7. If you must use an absolute **.org**, do not place it immediately after a branch instruction that has both a short and a long form.

Example:

```
@1000 0114        .org  $+100
                     # skip 100 decimal bytes (64 hex bytes)
                    .
                    .
@1000 0178     A:
```

See Also: .space

Chapter 2

.set

Purpose: Sets the label **name** equal to the expression **exp**, both in value and in type.

Format: .set name,exp

Remarks:

- Forward references are allowed within a module. That is, you can use **name** before you define it in a **.set**.
- **Exp** cannot be an undefined external expression.
- **Exp** can refer to a register number, but cannot refer to the contents of a register at runtime.
- Using **.set** may help to avoid errors if you have a frequently used expression. Equate the expression to a symbol, then refer to the symbol rather than the expression. If you need to change the value of the expression, you will only do so within the **.set** statement. However, you will then need to reassemble, since **.set** assignments occur at assembly time.

Examples: *Example 1*

```
instr: balr 12,4
              .
              .
     .set begin,instr
         # begin and instr are both relocatable
```

Example 2

```
.set ap,14      # assembler assigns value 14 to
                # the symbol ap -- ap is absolute
             .
             .
 lis ap,2       # assembler substitutes value 14 for the symbol ap
                # note that ap is a register
                # number in context as lis operand
```

Example 3

```
.set expr,A-(B+C)/(33*D)
```

See Also: "Symbols" on page 2-13

.short

Purpose: Assemble expressions **exp** into consecutive halfwords.

Format: .short exp,exp, ...

Remarks:

- Halfword alignment occurs as necessary.
- The number of expressions is limited by the amount of available memory.
- **Exp** cannot refer to the contents of any register.
- If **exp** is longer than a halfword, it is truncated.

Example:

```
.short 1,0x4444,fooble-333,0
```

Purpose: Skip **N** bytes in the output file and fill them with binary zeroes. **N** is an absolute expression.

The **.space** pseudo-op may be useful to reserve a chunk of storage in the data or text section of an assembler language program.

Format: .space N

Example:

```
@ 2000 0000 .space 444
     .
     .
@ 2000 01BC foo:
```

.text

Purpose: Switch to location counter **N** of the assembler language text section. **N** is an expression that evaluates to an integer from 0 to 3 inclusive.

Format: .text [N]

Remarks:

- The default **N** value is 0.
- The text section holds program instructions and read-only data.
- The assembler always begins in the text section, so you do not have to put **.text** at the beginning of a program.
- When the assembler first encounters **.text N**, text location counter **N** initially receives a value of 0. The assembler increments this value in bytes as instructions are read. If **.text N** occurs again with the same **N** value, that text location counter **N** keeps the same value it had the last time it was used.

Example:

```
.text 3
```

See Also: .comm, .data, .lcomm

Chapter 3

Purpose: Assigns **R** as the base register number. Bases relocatable expressions from register **R**, assuming that register **R** contains the relocatable program address of **exp** at runtime. **Exp** is a label or an expression involving a label. It represents the displacement or relative offset into the program, and must be relocatable.

Format: .using exp,R

Remarks:

- The **R** operand must be absolute and must evaluate to an integer from 0 to 15 inclusive.
- The **exp** operand cannot be or have an external or absolute symbol.
- With the information given in the **.using** pseudo-op, the assembler converts each relocatable expression (or implicit address) to a base register number plus a displacement. The linker later assigns the final addresses.
- The **.using** pseudo-op does *not* load the specified register; the programmer must guarantee that this value is actually in base register **R** at runtime.
- Symbol names do not have to be previously defined.
- The **.using** pseudo-op only affects instructions with based addresses (that is, the loads and stores).

Example:

```
.using _subrA,1
 # r1 can now be used for addressing with
 # displacements calculated relative to _subrA
  .
  .
.drop 1
 # stop using r1
.using _subrB,1
 # now assembler calculates displacements
 # relative to _subrB
```

See Also: .drop

Chapter 3

.xaddr

Purpose: Reserves **R** as the register to use for extended addressing. **R** must be an integer from 0 to 15 inclusive. If **R** is zero, extended addressing is disabled.

Extended addressing allows a displacement of more than 15 bits (a decimal value of +32,767 to -32,768 bytes) for the following instructions:

cal
ior
iow
l
lc
lh
lha
lm
lps
st
stc
sth
stm
tsh

These instructions have displacement and base register operands of the form **D2(R2)**. For extended addressing, any labels used as part of the displacement must be previously defined. Also, the displacement cannot be calculated with the **.using** pseudo-op.

When the assembler encounters one of the instructions listed above, the assembler pads the instruction's displacement **D2** with leading zeroes until the displacement is a 32-bit value. From this 32-bit displacement, the assembler calculates two values:

L = The low-order 15 bits of the 32-bit displacement
H = The high-order 16 bits of the 32-bit displacement, plus the highest-order bit of the low-order halfword of the 32-bit displacement.

The assembler then examines the **H** value:

- If **H** = 0, the assembler does not change the originally specified instruction. This implies a no-op instruction.
- If **H** ≠ 0 and the **.xaddr**'s register **R** is 0, the assembler generates a `Displacement too large` error message.
- If **H** ≠ 0 and the **.xaddr**'s register **R** is valid, then the assembler generates the following code. Assume that the instruction covered by **.xaddr** is ***op R1,D2(R2)***, where *op* is one of the instructions listed above:

```
cau  R,H(R2)     # R is the .xaddr operand
                 # R2 is the base register for instruction op
op   R1,L(R)     # R1 is the source or target register
                 # specified for instruction op
                 # R is the .xaddr operand
```

Format: .xaddr R

Remarks:

- Instructions with extended addresses may follow branch-with-execute instructions (for example, **bnbx**), as long as the branch-with-execute instruction does not refer to **.xaddr**'s register **R**. The assembler inserts a **cau** instruction, then the branch-with-execute instruction, then the altered instruction. (See Example 2.)
- The **-a** flag causes the C compiler to generate **.xaddr** pseudo-ops.
- The **.xaddr** pseudo-op does not save the contents of register **R**. The program must explicitly save and restore the contents of that register.
- The **.xaddr** pseudo-op does not affect the following instructions:

  aei
  cal16
  cau
  ci
  cli
  nilo
  nilz
  niuo
  niuz
  oil
  oiu
  sfi
  svc
  ti
  xil
  xiu

Examples:

Example 1

```
.xaddr 7
st     8,0x854003(1)
```

This causes the assembler to generate the following:

```
    cau 7,0x85+0(1)  # displacement for cau is high half of displacement
                     # from original store instruction, plus high-order
                     # bit of lower half of displacement from original
                     # store instruction
    st  8,0x4003(7)
```

Example 2

```
.xaddr 7
brx    5
l      4,0x7FFFE(1)
```

This causes the assembler to generate the following:

```
    cau 7,7+1(1)    # displacement for cau is high half of displacement
                    # from original load instruction, plus high-order
                    # bit of lower half of displacement from original
                    # load instruction
    brx 5
    l   4,0x7FFE(7)
```

Chapter 6. Assembling, Linking and Running a Program

CONTENTS

About This Chapter

At this point, you have created an assembler language program with the editor of your choice. The program may consist of several modules or files. You may have included macros with **m4**, or subroutines such as **clock** or the floating point subroutines.

After writing your assembler language program, you need to assemble each file and link the files together. There are two ways to do this:

1. Use the **as** command to assemble each file and the **ld** command to link the files, or
2. Use the **cc** command to assemble and link the files in a single step. You may prefer to use this method, because **cc** automatically calls **ld** with the flags necessary to link assembler language source files.

After the files have been linked by either method, you can run the program.

This chapter explains the AIX Operating System commands needed to assemble, link, and run your AIX Operating System assembler language program. This chapter also includes information you need to link your file with AIX Operating System system calls or with files written in other languages.

Note: If you want your program to include information for debuggers such as **sdb**, you should follow the conventions in "Subroutine Linkage and System Calls" on page 6-10, even if your program does not call any routines. Also be sure to include the information shown in "Traceback" on page 6-21.

Assembling and Linking with cc

By default, the **cc** command starts the AIX Operating System C preprocessor, the C compiler, the 032 Microprocessor assembler, and the link editor. If you specify the **-O** flag, **cc** also starts the optimizer. The **cc** command is normally used with C language source code. However, **cc** can also be used with AIX Operating System assembler language files as input files. If you run **cc** on an assembler language file or files, **cc** automatically runs **as** and **ld** on the files (unless you specify the **-c** flag). Furthermore, **cc** automatically does the following:

- Calls **ld** with the **-T0x10000000, -n, -lc, -lrts,** and **-K** flags
- Links the **/lib/crt0.o** file
- Specifies **ld -estart** to name the executable output file's entry point (where the label "start" resides in **/lib/crt0.o**).
- Searches library files **libc.a** and **librts.a**.

All of these actions are required for assembler source files. The **cc** command can thus be used to assemble and link assembler language source code in a single step.

A few of the flags available for **cc** are shown below. See *AIX Operating System Commands Reference* for a complete explanation of the **cc** command.

cc [**-c**] [**-o** *file*] [**-O**] [**-a**] *file* . . .

Flag **Purpose**

file If the input *file* is an assembler language source program, the file name must end in **.s**. The **cc** command calls **as** for all the **.s** files, then **cc** calls **ld**. After **ld** is finished, **cc** deletes the **.o** file if you specified a single file. If you specified multiple files, the **.o** files will not be deleted. The output file will be named **a.out** by default.

-c Do not run **ld** on the completed object files. Leave the output as object code in the files with **.o** suffixes. Using this flag on assembler language source code is equivalent to running **as**.

-o *file* This allows you to specify the output *file* name instead of the default **a.out**. This flag is ignored if you specify the **-c** flag.

-O Invokes the optimizer.

-a Enables extended addressing. You should use this flag if a compiled procedure creates a stack greater than 32,767 bytes. This flag causes the compiler to reserve a register for use by the assembler. Therefore, this flag reduces the number of available register variables by one. (If you use this flag when your procedure does not require extended addressing, the number of the procedure's register variables will be needlessly reduced.) For more information, see the **.xaddr** pseudo-op in Chapter 5.

Note: Besides the **ld** flags that **cc** automatically calls, you may also insert other **ld** flags on the command line with **cc**. The **cc** command will pass the flags to **ld**.

Assembling and Linking with Two Separate Steps

You may choose to assemble your modules with the **as** command, then link them with the **ld** command.

Assembling with as

The **as** command causes the assembler to process a single file of assembler language source code. See "The Assembler's First and Second Passes" on page 6-6 for more information. If you use **as**, you must later use **ld** as well.

As assembles a file to produce assembler object code:

as [**-o** *objfile*] [*file*]

The *file* is the input file containing assembler language source code. By default, the assembler assumes that the input comes from standard input. If the input file came from a compiler or if the input file was written in assembler language, the filename ends with **.s**. If the input file cannot be read, the assembly will terminate with the message "Cannot open *file*."

The **-o** *objfile* flag puts the assembler output into *objfile*. By default, the output file is named **a.out**.

If you want to use macros in your assembler language source file, you must invoke **m4** explicitly. See *AIX Operating System Programming Tools and Interfaces* for information on **m4**.

The output file is ready to link if no errors occurred and if there are no unintended external references. If the assembler detects errors, it writes the following information to standard error:

- Input file name
- Line number where the error occurred in the assembly source code
- A descriptive message of the problem.

If the **as** command produces any error messages, you should correct the errors before moving on. See *Messages Reference* for assembly error messages.

The Assembler′s First and Second Passes

When you enter the **as** command, the assembler makes two passes over the source program.

The First Pass

On the first pass, the assembler:

1. Allocates space for instructions and storage areas you requested
2. Where possible, fills in values of constants
3. Builds a symbol table, also called a cross-reference table, and makes an entry in this table for every symbol it encounters in the label field of a statement.

The assembler reads one source statement at a time. If the source statement has a valid symbol in the label field, the assembler checks that the symbol has not already been used as a label. If this is the first time the symbol has been used as a label, the assembler adds the label to the symbol table and assigns the value of the current location counter to the symbol. If the symbol has already been used as a label, the assembler gives the error message "Redefinition of *symbol*" and re-assigns the symbol value.

Next, the assembler examines the instruction′s mnemonic. If the mnemonic is for a machine instruction, the assembler determines the format of the instruction (for example, BI format). Then the assembler allocates the number of bytes necessary to hold the machine code for the instruction. The contents of the location counter are incremented by this number of bytes.

When the assembler encounters a comment (#) or an end-of-line character, the assembler starts scanning the next instruction statement. The assembler keeps scanning statements and building its symbol table until there are no more statements to read.

At the end of the first pass, all the necessary space has been allocated, and each symbol defined in the program is associated with a location counter value in the symbol table. When there are no more source statements to read, the assembler chooses short or long forms of the branch instructions, then the second pass starts at the beginning of the program again.

The Second Pass

On the second pass, the assembler:

1. Examines operands for symbolic references to storage locations, and resolves these symbolic references by referring to the information in the symbol table
2. Translates source statements into machine code and constants, thus filling the allocated space with object code
3. Produces a file containing error messages, if any.

At the beginning of the second pass, the assembler scans each source statement a second time. As the assembler translates each instruction, it increments the value contained in the location counter.

If a particular symbol appears in the source code but is not found in the symbol table, then the symbol was never defined. That is, the assembler did not encounter the symbol in the label field of any of the statements scanned during the first pass, or was never the subject of a **.set**, **.lcomm** or **.comm**.

This could be either a deliberate external reference or an accidental programmer's error (such as misspelling the symbol name). The assembler cannot determine whether this is a true error, so it does not send any error message. Also, if the symbol is preceded by a **.globl**, **as** assumes the symbol is defined externally and will not give you a message.

The assembler logs errors such as incorrect data alignment. Assembler error messages are shown in the alphabetic section of *Messages Reference.*

After the programmer corrects assembly errors, the program is ready to be linked.

Linking with ld

You should use the **ld** command if you have previously used **as** or **cc -c** to assemble a file.

The **ld** command performs the functions of a link editor. It combines several object files, relocates them, resolves external symbols, searches libraries, and gives symbol table information to **sdb**. The object files can be from a high-level language, or assembler language, or both. The final result is an executable object module which is named **a.out** by default. This object module has a fixed format which is not affected by **ld** flags.

Whenever **ld** links assembler language files, it puts assembler sections together in the proper order, even if parts of any section came from different files.

There are several things you must always do when linking assembler language files with **ld**:

- Specify the **-T0x10000000** flag so that your non-kernel assembler language programs will be linked at hex location 10000000 or higher.
- Specify the **-estart** flag to make `start` the executable output file's entry point. The label `start` is the point in **/lib/crt0.o** that will first get control from the kernel.
- Specify the **-n** flag to make read-only program text that is shared among all users running the file.
- Specify the **-K** flag to enable mapped files.
- Specify the **/lib/crt0.o** file along with the assembler language file or files on the command line. The kernel interfaces with the **/lib/crt0.o** routine to set up the required registers and to otherwise define the calling sequence before executing code. The beginning of this routine is a label named `start`.
- Specify the **-l** flag to search the appropriate library file. For example, specifying **-lc** will search **libc.a**, the standard system library for C and assembler language programs.

If you use **cc** to assemble and link your assembler language files, **cc** will do these things automatically. (The **/etc/cc.cfg** file contains information on the **ld** flags that **cc** uses.)

The **ld** command maps all common segments with the same name into the same memory. (See the **.globl** pseudo-op in Chapter 5.)

A few of the flags available on **ld** are shown below. See *AIX Operating System Commands Reference* for a complete list of **ld** flags. Also see page 5-27 to learn about special **ld** considerations for assembler files that use absolute **.orgs**.

ld [**-n**] [**-K**] [**-s**] [**-x**] [**-X**] [**-o***name*] [**-e***label*] [**-T***num*] [**-l***key*] *file* . . .

Flag	**Purpose**
file...	Link the specified file or files. One of these should be **/lib/crt0.o**.
-n	Make the text segment read-only and shared among all users running the file.
-K	Load the **a.out** header into the first bytes of the text segment, followed by the text segments from the object modules. This causes pages of executable files to be aligned on pages in the file system.

-s Remove symbol table and relocation bits. You can use **sdb** but you cannot use symbol names in **sdb** in the executable module. This flag is automatically turned off if there are any undefined symbols.

-x Do not preserve local (non-**.globl**) symbols in the output symbol table; only enter external symbols. This flag saves some space in the output file but allows **sdb** to reference external symbols by name.

-X Save local symbols except for those whose names begin with "L." This flag is used to discard labels generated internally by the compiler while retaining symbols local to routines. This flag saves some space in the output file but does not impair the usefulness of **sdb**.

-o*name* Use *name* as the name of the **ld** output file instead of **a.out**.

-e*label* Use the location in the output file named by *label* as the entry point when the program is executed. Whenever you link the **/lib/crt0.o** file (that is, when you link assembler language files), you must always specify the *label* as `start`.

Note: The **cc** command calls **ld** with **-estart** by default.

-T*num* Make *num* the starting address for the output file's text segment. This specifies an absolute origin. By default, **ld** assumes that object modules are position-independent. Also, by default, the text segment begins at location zero.

Note: The **cc** command calls **ld** with **-T0x10000000** by default.

-l Searches the specified library file, where *key* selects the file **lib***key***.a**. If you specify **-lc** or **-l** with no *key*, **ld** chooses **libc.a**, the standard system library for C and assembler language programs.

Note: The **cc** command calls **ld** with **-lc** by default.

After files have been linked, any symbol definition should be found in the same **a.out** segment as all references to that symbol. Also, at linkage, all references to external symbols should have been resolved. If they are not resolved, **ld** will give you an error message. The only exception is for symbols that are part of an absolute expression used in an **.org** pseudo-op.

By default, errors produced during the **ld** command go to standard error. After the programmer corrects **ld** errors, the program is ready to run.

Subroutine Linkage and System Calls

The object format calling conventions shown below are used by the C language. You must follow these conventions if you want to link from an assembler language file to a C language routine or if you want to make AIX Operating System kernel calls from within an assembler language program. If you want to link to a routine in another language, you must use the conventions for that language.

The symbolic debugger, **sdb**, will give you information about the following items whether you use the C language calling conventions or not:

- Registers
- Absolute addresses
- Global symbol names defined in your program.

If you use the C language calling conventions described in this section, however, **sdb** will also let you look at stack frames. If you do not use the C language linkage conventions, you cannot look at stack frames.

The runtime segment layout is fixed, with separate segments allocated for sharable text, private data, and the stack. By convention, segment register 1 is for program text, segment register 2 is for static data, and segment register 3 is the process stack.

Register Usage

To be compatible with C language, called and calling routines must observe certain conventions on register usage. In the discussion that follows, a ***volatile*** register refers to a register whose value on entry need not be preserved when the called routine returns. For a ***non-volatile*** register, however, the register's value on entry must be preserved on exit from the called routine. If the value of a non-volatile register changes across the call, then the called routine must do the following:

- Save the value of the register before the register is changed
- Restore the original value of the register before returning.

C-compatible routines must use registers as follows:

- Register 1 (the stack frame pointer) is non-volatile.
- Registers 6 through 14 inclusive are non-volatile.
- Floating point registers 2 through 5 inclusive are non-volatile.
- All other registers are volatile.

Notes:

1. C routine prologs save the constant pool pointer in register 14. However, the calling sequence does not require this.
2. The Multiplier Quotient (MQ) register is not preserved across calls.
3. You can use the **.set** pseudo-op to define the names of registers. For example,

   ```
   .set fp,1
   ```

   allows you to use the symbol "fp" to refer to GPR 1.
4. You may find the **lm** and **stm** instructions useful to load or store several registers at once. See Chapter 4 for more information about these instructions.

On entry, the registers have the values shown in Figure 6-1.

Register	Name	Use
0	called pcp	Constant pool pointer for called routine
1	fp	Caller's stack frame pointer
2	P1	First word of parameters (if parameters are passed)
3	P2	Second word of parameters (if needed)
4	P3	Third word of parameters (if needed)
5	P4	Fourth word of parameters (if needed)
6	--	Data local to caller
7	--	Data local to caller
8	--	Data local to caller
9	--	Data local to caller
10	--	Data local to caller
11	--	Data local to caller
12	--	Data local to caller
13	--	Data local to caller
14	--	Data local to caller
15	link	Return address

Figure 6-1. Register values on subroutine entry

On exit, the registers have the values shown in Figure 6-2.

Register	Name	Use
0	--	Undefined
1	fp	Caller's stack frame pointer
2	--	Routine's returned value, if returned value is 32 bits or less; or, high-order 32 bits of floating point value; otherwise undefined
3	--	Low-order 32 bits of floating point value; otherwise undefined
4	--	Undefined
5	--	Undefined
6	--	Value of caller's register 6 on entry
7	--	Value of caller's register 7 on entry
8	--	Value of caller's register 8 on entry
9	--	Value of caller's register 9 on entry
10	--	Value of caller's register 10 on entry
11	--	Value of caller's register 11 on entry
12	--	Value of caller's register 12 on entry
13	--	Value of caller's register 13 on entry
14	--	Value of caller's register 14 on entry
15	--	Undefined

Figure 6-2. Register values on subroutine exit

The Stack Frame

When an assembler language program calls another routine or the kernel, arguments are passed on the stack. The stack grows from higher addresses to lower addresses. Note that temporary variables are allocated in the frame, not at the top of the stack. A single frame pointer register is used to address local storage for a routine, incoming and outgoing arguments, and the save area.

Everything in the stack is aligned on word boundaries. The length of each area defined in the stack is an integer number of words.

Figure 6-3 on page 6-15 represents the contents of a stack frame. In this figure, the current routine has acquired a stack frame which allows it to call other functions. If no functions are called and there are no local variables, then the function need not allocate a stack frame or adjust the frame pointer. It can still use the register save area at the bottom of the caller's stack frame, if needed. It can also use up to 256 bytes below the frame pointer for any use (for example, saving floating point registers 2 through 5 if they are changed).

The first four words of arguments are passed in registers 2 through 5. Any other arguments are passed on the stack; the frame pointer points directly to these arguments. The called routine can save its first four parameters in the four words of memory immediately below the stack frame pointer. This allows the entire parameter area to be accessed as a storage array.

Note that since the frame size is known, input parameters can be addressed using the current stack pointer as a base register.

Each stack frame has one five-word area that is reserved for future system use. Your programs should never modify these reserved areas.

The Stack Floor

The stack floor is defined as **-256(1)**, where the contents of register 1 point into the segment addressed by segment register 3. That is, segment register 3 contains the stack segment identifier. Some other segment register may be used for the stack, but the kernel will not cause such a stack to grow. In this case, the application is responsible for maintaining the stack size; failure to do so results in an addressing exception.

All programs in the system must avoid accessing locations in the stack segment that are below the stack floor. The AIX Operating System kernel will extend the stack segment on a page fault if the following conditions are met:

- The faulting address is not below the stack floor (that is, on top of the stack).
- The stack limit has not been exceeded.
- Segment register 3 points to the stack.

HIGH ADDRESSES

	caller's stack area	
	input Pn	The nth or last word of *input* parameters being passed to this routine.
	• • • •	From the current routine's point of view, the input parameter area holds parameters passed from the caller. P5 through Pn are passed in storage.
caller's stack frame pointer →	input P5	The fifth word of input parameters being passed to this routine.
	input P4	If desired, current routine may store contents of register 5 (fourth word of parameters) here.
	input P3	If desired, current routine may store contents of register 4 (third word of parameters) here.
	input P2	If desired, current routine may store contents of register 3 (second word of parameters) here.
	input P1	If desired, current routine may store contents of register 2 (first word of parameters) here.
	reserved	5 words.
	current save area	Caller's registers are saved here. Register 15 is highest word of save area.
	floating point register save area	If required, allocated by called routine as needed.
	local stack area	Most automatic variables and temporaries can be addressed using short loads and stores.
	output Pmax • • • •	If there are >4 words of parameters passed, Pmax is the number of words required to call the routine that has the most words of parameters. That is, Pmax is the greatest number of parameter words the current routine will ever use to call a subroutine. The current routine will reserve this space when updating its frame pointer. From the current routine's point of view, the *output* parameter area is used for passing parameters to functions that the current routine, in turn, calls.
current stack frame pointer →	output P5	If there are >4 words of parameters passed, this is the fifth word of input parameters passed to the next called routine.
	output P4	Routines you call may save their fourth word of parameters here. This word is always reserved, whether this routine calls others or not.
	output P3	Routines you call may save their third word of parameters here. This word is always reserved, whether this routine calls others or not.
	output P2	Routines you call may save their second word of parameters here. This word is always reserved, whether this routine calls others or not.
	output P1	Routines you call may save their first word of parameters here. This word is always reserved, whether this routine calls others or not.
	reserved	5 words.
	next save area	Functions you call will save your registers here. They will determine the size (up to a maximum of 64 bytes).
stack floor →	floating point register save area	Floating point registers 5 through 2 saved in that order (up to a maximum of 64 bytes).

LOW ADDRESSES

Figure 6-3. Contents of a Stack Frame

When a signal is received, the signal handler is pushed onto the existing stack, so any data below the stack floor is subject to destruction without notice.

All compilers and user-written assembler routines must maintain the following system invariants:

- The stack pointer is always valid (unless interrupts are disabled).
- Data is never saved or accessed below -256(1). Register 1 is the stack frame pointer, and the negative displacement must be large enough to allow access to save areas for both general purpose registers and floating point registers.
- When the stack frame size is more than 32767 bytes, it cannot be incremented or decremented in a single instruction. In this case, the stack frame size must decrease and increase monotonically. That is, the value must increase continuously without any intermediate decreasing values, or decrease continuously without any intermediate increasing values. This ensures that there is no timing window in which a signal handler would either overlay valid stack data, or erroneously appear to overflow the stack segment.

The Constant Pool

Each routine uses its own pool of constants and pointers. At call time, the routine's caller must pass the address of the constant pool—that is, the pcp—in register 0. Ordinarily, the called routine will save the caller's registers, then copy this address to another register (C language uses register 14).

The first word in a routine's constant pool must contain the address of the beginning of the code for the routine (that is, the routine's entry point). When used as a C language function pointer, the "address" of the routine is in fact the address of its constant pool.

All addresses of external routines end up in the constant pool of each routine that calls them. These addresses are 32-bit values.

The Calling Routine's Responsibilities

When an assembler language program calls another program, the caller should not use the names of the called program's commands, functions, or procedures as global assembler language symbols. To avoid confusion, you may want to remember the following C language conventions when you create symbol names.

By convention, for a C language routine named **foo**, the C compiler assigns **_foo** as the external name of the constant pool, and **.foo** as the external name of its entry point. For example, suppose the C compiler processes the following C language program:

```
main()
{
       int a;
       a = somfunc(1, 2, 3);
}
```

The assembler language code produced by the compiler includes the following lines:

```
        .text
        .align  1
        .globl  .main
.main:                 # This is the external name of the first
                       # executable piece of code.  Also,
                       # this is the beginning of the code.
            .
            .
        lis    2,1     # parameter 1 for somfunc
        lis    3,2     # parameter 2 for somfunc
        lis    4,3     # parameter 3 for somfunc
        call   .somfunc, 4(14),3
                       # this calls somfunc
        st     2,0(1)  # save the return value in "a"
            .
            .
        .data   3          # constant pool
        .globl  _main      # external function name
_main:  .long   .main      # address of first instruction
        .long   _somfunc   # address of somfunc constant pool
```

To avoid confusion, you might decide that your own local symbols should not begin with an underscore or period. However, when you are linking an assembler routine with a C language program, the C compiler references the starting address of the assembler routine's constant pool, and the starting address of the assembler routine's code. In this case, the assembler symbol name for that constant pool should start with an underscore, and the name of the assembler code's starting address should start with a period, so that the C compiler will recognize them.

The calling routine must reserve space in its local frame for the largest argument list it will require. If it never calls a routine with more than four words of parameters, no space need be reserved.

The calling routine provides space for the called routine's register save. At call time, the calling routine is responsible for putting the pcp (the address of the constant pool) into the called routine's register 0. (The **call** and **callr** pseudo-ops do these things automatically.) If necessary, the calling routine should save and restore its own registers 0, 2, 3, 4, 5, and 15. The C compiler generates code to move the contents of register 0 to register 14 on entry.

If it knows the called routine's name, the caller uses the code shown in Figure 6-4 at a function call.

```
l       2,P1            # Load the first word of parameters into register 2
l       3,P2            # Load the second word of parameters into register 3
l       4,P3            # Load the third word of parameters into register 4
l       5,P4            # Load the fourth word of parameters into register 5

st      P5,4*(5-5)(1)   # Store the fifth word of parameters
                        # (if it exists) on the stack
        .
        .
st      Pn,4*(n-5)(1)   # Store the nth word of parameters
                        # (if it exists) on the stack

call    .name,pointer,number_words
                        # Call the routine.
                        # .name is entry point of called routine.
                        # pointer is the address of the called
                        # routine's pcp -- that is, 12(14).
                        # number_words is the number of words
                        # of parameters that are being passed.
```

Figure 6-4. What a Calling Routine Does. P1 is the first word of parameters, P2 is the second word of parameters, and so on. The assembler expands **call** as shown on page 5-14. For clarity, this sample code shows the operands being evaluated in order. However, this is not required. It is usually more convenient to evaluate the first four parameter words last, since they tie up several registers.

If a function pointer is used, the caller will not know the name of the called routine. In this case, the calling routine uses the code shown in Figure 6-5.

```
l       2,P1             # Load the first word of parameters into register 2
l       3,P2             # Load the second word of parameters into register 3
l       4,P3             # Load the third word of parameters into register 4
l       5,P4             # Load the fourth word of parameters into register 5

st      P5,4*(5-5)(1)    # Store the fifth word of parameters
                         # (if it exists) on the stack
        .
        .
st      Pn,4*(n-5)(1)    # Store the nth word of parameters
                         # (if it exists) on the stack

l       15,_routine      # get the routine's pcp

callr   15,number_words # call the routine
```

Figure 6-5. What a Calling Routine Does When Using Function Pointer. P1 is the first word of parameters, P2 is the second word of parameters, and so on. The assembler expands **callr** as shown on page 5-16.

Note: To make system calls inside an AIX Operating System assembler language program, use the interface routines in the **libc.a** library file. You can access this file automatically with the **cc** command, or explicitly with the **ld** command.

The Called Routine's Responsibilities

A called routine typically responds as shown in Figure 6-6.

```
routine:           # Function entry

stm  n,-regoff(1)  # Save registers n through 15.
                   # n is the first register modified above register 5.
                   # regoff = 4 * (16-n) + (4 * 9)
                   # -- 4 words to save first 4 words of
                   # parameters, and 5 reserved words.

cal  1,-fsize(1)   # Increases the stack segment by fsize bytes
                   # -- only if stack space needed.
                   # fsize = regoff + 4*(max(0,Pmax-4)) + localsize
                   # Where Pmax is maximum number of words of parameters
                   # passed to other routines, localsize is size of your
                   # local stack area.  This line is required if the called
                   # routine needs space on the stack for local automatic
                   # variables, or if it calls another routine
                   # (since that would require a new save area).

lr  14,0           # Save pcp in preserved register
lr  13,2           # Save first parameter in a register
st  3,fsize-8(1)   # Save second parameter on the stack
  .
  .
  .
# Return sequence, assume register 2 is set

lm   n,fsize-regoff(1) # Restore registers that were saved
                   # (registers n through 15)
brx  15            # Return to the point called
cal  1,fsize(1)    # Restore the caller's stack pointer
```

Figure 6-6. What a Called Routine Does.

The called routine may assume that registers 0, 1, and 15 are set as specified in "The Calling Routine's Responsibilities" on page 6-17. Other register use is conventional but not mandatory. If the called routine changes any of the general purpose registers 6 through 14 or the floating point registers 2 through 5, it must store those changed non-volatile registers at standard locations below the caller's stack frame pointer, as the following table shows.

Register	Stored at
15	-40(1)
14	-44(1), if saved
13	-48(1), if saved
12	-52(1), if saved
11	-56(1), if saved
10	-60(1), if saved
9	-64(1), if saved
8	-68(1), if saved
7	-72(1), if saved
6	-76(1), if saved

Floating point registers 5 through 2 are saved, in that order, immediately below the lowest GPR saved. The called routine only needs to leave space for registers that are actually being saved. If a register is not being saved, the called routine should not leave space for it.

Traceback

The called routine does not have to save its caller's stack pointer, because the called routine knows how much the stack pointer has been decremented. However, debugging aids such as **sdb** need to be able to unravel the call/return stack.

Each module has a traceback table in the text segment at the end of its code. This table can be found by scanning forward from the IAR at the point of interruption. The beginning of this table is marked by two consecutive halfwords, each aligned on a halfword boundary. The first byte of each halfword has the value 0xDF. These decode as invalid instructions, so they will not be present in normal code.

The traceback table is described in the file **/usr/include/sys/debug.h**. The traceback table has the following information, in order:

- 1 byte with the value 0xDF.
- 1 byte of a code field. The values in this field are:

  0 -- Unknown type of stack frame.
  1 -- Normal C-like debug data.
  2 -- Assembler routine with no stack frame.
  3 -- Debug tag for **/lib/crt0.o**. This is the bottom of the stack.
  4 -- Stack frame was generated by a signal.
  5 -- Assembler routine with no parameters or frame.
  6 -- C-like debug data, if floating point registers were saved or if the frame size was greater than 0x7FFF.

  All other values are reserved.
- 1 byte with the value 0xDF.

- 1 byte of flags. If the code field had a value ≠ 1 or ≠ 6, the flag value is undefined. If the code field had a value of 1 or 6, the high-order four bits of this byte show the lowest register number that was saved on procedure entry. The low-order four bits indicate which of the first four parameter words are register variables. (This information is used only by debuggers such as **sdb**.) These low-order four bits have the following meaning:

  highest-order bit -- Has value of 1 if the parameter passed in register 5 is a register variable.
  next bit -- Has value of 1 if the parameter passed in register 4 is a register variable.
  next bit -- Has value of 1 if the parameter passed in register 3 is a register variable.
  lowest-order bit -- Has value of 1 if the parameter passed in register 2 is a register variable.

- If the code field had a value of 1, a halfword containing the size of this procedure's stack frame. If the code field had a value of 6, a full word containing the size of this procedure's stack frame, where the high-order four bits of the word represent the lowest floating point register pair that was saved. If the code field had a value other than 1 or 6, this halfword or word is undefined.

032 Microprocessor assembler language programs should include traceback information to help debuggers such as **sdb**. The following examples show how to code traceback information in 032 Microprocessor assembler language.

Example 1

An assembler language program may be entirely self-contained, and may not call any other routines. Such programs do not allocate their own stack frame. The text section of such a program should have the following lines of code at the end of the executable instructions:

```
.short  0xDF02  # code field 02 means no stack frame
.short  0xDF00  # flag value 00 undefined
```

Example 2

An assembler language program may call other routines and allocate its own stack frame. For example, suppose we have such a program that allocates a 128-byte stack frame, saves registers 10 through 15 on entry, and keeps all its parameters in non-volatile registers 13 through 10. This program would have the following lines of code at the end of its text section:

```
.short  0xDF01  # code field 01 means normal debug data
.short  0xDFAF  # flag value AF means register 10 was lowest
               # register saved on procedure entry; parameters passed
               # in registers 2 through 5 were register variables
               # and are in registers 13, 12, 11, and 10 respectively
.set    fsize,128    # frame size
.short  fsize        # amount register 1 decremented
```

Running the Program

Your program is ready to run when it has been assembled and linked without producing any error messages. To run a program, first be sure you have AIX Operating System permission to execute the file. Then, simply type the program's name at the AIX Operating System prompt:

```
$ filename
```

By default, the output of the program goes to the standard output. To direct output to a place other than the standard output, use the AIX Operating System shell > operator. See *Using and Managing the AIX Operating System* for more information on AIX Operating System shell commands.

You can diagnose runtime errors by invoking the symbolic debugger with the AIX Operating System **sdb** command. The symbolic debugger works on any code that follows C language calling conventions. All compiler-generated code, but not necessarily all human-generated code, can be used with **sdb**. For more information about the symbolic debugger, see *AIX Operating System Programming Tools and Interfaces.*

Appendix A. Mnemonic and Op Code Tables

This appendix consists of two tables. The first table lists the instructions alphabetically by mnemonic. The second table lists the instructions numerically by op code.

Instructions, Indexed by Mnemonic

Op codes are represented with hexadecimal values. Where more than one op code is shown for a mnemonic, the assembler automatically chooses the correct op code.

Mnemonic	*Operands*	*Op Code*	*Instruction Name*	*CS Bits Changed*
a	R1,R2	E1	Add	LT, EQ, GT, C0, OV
abs	R1,R2	E0	Absolute	LT, EQ, GT, C0, OV
ae	R1,R2	F1	Add Extended	LT, EQ, GT, C0, OV
aei	R1,R2,I3	D1	Add Extended Immediate	LT, EQ, GT, C0, OV
ai	R1,R2,I3	C1	Add Immediate	LT, EQ, GT, C0, OV
ais	R1,I2	90	Add Immediate Short	LT, EQ, GT, C0, OV
b [1]	A1	88	Branch (long)	-- none --
		0	Branch (short)	-- none --
bala	A1 or I1	8A	Branch and Link Absolute	-- none --
balax	A1 or I1	8B	Branch and Link Absolute with Execute	-- none --
bali	R1,A2	8C	Branch and Link Immediate	-- none --
balix	R1,A2	8D	Branch and Link Immediate with Execute	-- none --
balr	R1,R2	EC	Branch and Link Using Register	-- none --
balrx	R1,R2	ED	Branch and Link Using Register with Execute	-- none --
bb [1]	I1,A2	8E	Branch on Condition Bit Immediate (long)	-- none --
		0	Branch on Condition Bit Immediate (short)	-- none --
bbr	I1,R2	EE	Branch on Condition Bit Immediate Using Register	-- none --
bbrx	I1,R2	EF	Branch on Condition Bit Immediate Using Register with Execute	-- none --
bbx	I1,A2	8F	Branch on Condition Bit Immediate with Execute	-- none --
bcc [1]	A1	88	Branch on Carry Bit Clear (long)	-- none --
		0	Branch on Carry Bit Clear (short)	-- none --
bccr	R1	E8	Branch on Carry Bit Clear Using Register	-- none --

Mnemonic	*Operands*	*Op Code*	*Instruction Name*	*CS Bits Changed*
bccrx	R1	E9	Branch on Carry Bit Clear Using Register with Execute	-- none --
bccx	A1	89	Branch on Carry Bit Clear with Execute	-- none --
bcs [1]	A1	8E	Branch on Carry Bit Set (long)	-- none --
		0	Branch on Carry Bit Set (short)	-- none --
bcsr	R1	EE	Branch on Carry Bit Set Using Register	-- none --
bcsrx	R1	EF	Branch on Carry Bit Set Using Register with Execute	-- none --
bcsx	A1	8F	Branch on Carry Bit Set with Execute	-- none --
beq [1]	A1	8E	Branch on Equal (long)	-- none --
		0	Branch on Equal (short)	-- none --
beqr	R1	EE	Branch on Equal Using Register	-- none --
beqrx	R1	EF	Branch on Equal Using Register with Execute	-- none --
beqx	A1	8F	Branch on Equal with Execute	-- none --
bge [1]	A1	88	Branch on Greater Than or Equal (long)	-- none --
		0	Branch on Greater Than or Equal (short)	-- none --
bger	R1	E8	Branch on Greater Than or Equal Using Register	-- none --
bgerx	R1	E9	Branch on Greater Than or Equal Using Register with Execute	-- none --
bgex	A1	89	Branch on Greater Than or Equal with Execute	-- none --
bgt [1]	A1	8E	Branch on Greater Than (long)	-- none --
		0	Branch on Greater Than (short)	-- none --
bgtr	R1	EE	Branch on Greater Than Using Register	-- none --
bgtrx	R1	EF	Branch on Greater Than Using Register with Execute	-- none --
bgtx	A1	8F	Branch on Greater Than with Execute	-- none --
ble [1]	A1	88	Branch on Less Than or Equal (long)	-- none --
		0	Branch on Less Than or Equal (short)	-- none --
bler	R1	E8	Branch on Less Than or Equal Using Register	-- none --

Mnemonic	*Operands*	*Op Code*	*Instruction Name*	*CS Bits Changed*
blerx	R1	E9	Branch on Less Than or Equal Using Register with Execute	-- none --
blex	A1	89	Branch on Less Than or Equal with Execute	-- none --
blt [1]	A1	8E	Branch on Less Than (long)	-- none --
		0	Branch on Less Than (short)	-- none --
bltr	R1	EE	Branch on Less Than Using Register	-- none --
bltrx	R1	EF	Branch on Less Than Using Register with Execute	-- none --
bltx	A1	8F	Branch on Less Than with Execute	-- none --
bnb [1]	I1,A2	88	Branch on Not Condition Bit Immediate (long)	-- none --
		0	Branch on Not Condition Bit Immediate (short)	-- none --
bnbr	I1,R2	E8	Branch on Not Condition Bit Immediate Using Register	-- none --
bnbrx	I1,R2	E9	Branch on Not Condition Bit Immediate Using Register with Execute	-- none --
bnbx	I1,A2	89	Branch on Not Condition Bit Immediate with Execute	-- none --
bne [1]	A1	88	Branch on Not Equal (long)	-- none --
		0	Branch on Not Equal (short)	-- none --
bner	R1	E8	Branch on Not Equal Using Register	-- none --
bnerx	R1	E9	Branch on Not Equal Using Register with Execute	-- none --
bnex	A1	89	Branch on Not Equal with Execute	-- none --
br	R1	E8	Branch Using Register	-- none --
brx	R1	E9	Branch Using Register with Execute	-- none --
btc [1]	A1	88	Branch on Test Bit Clear (long)	-- none --
		0	Branch on Test Bit Clear (short)	-- none --
btcr	R1	E8	Branch on Test Bit Clear Using Register	-- none --
btcrx	R1	E9	Branch on Test Bit Clear Using Register with Execute	-- none --
btcx	A1	89	Branch on Test Bit Clear with Execute	-- none --
bts [1]	A1	8E	Branch on Test Bit Set (long)	-- none --
		0	Branch on Test Bit Set (short)	-- none --
btsr	R1	EE	Branch on Test Bit Set Using Register	-- none --

Mnemonic	*Operands*	*Op Code*	*Instruction Name*	*CS Bits Changed*
btsrx	R1	EF	Branch on Test Bit Set Using Register with Execute	-- none --
btsx	A1	8F	Branch on Test Bit Set with Execute	-- none --
bvc [1]	A1	88	Branch on Overflow Clear (long)	-- none --
		0	Branch on Overflow Clear (short)	-- none --
bvcr	R1	E8	Branch on Overflow Clear Using Register	-- none --
bvcrx	R1	E9	Branch on Overflow Clear Using Register with Execute	-- none --
bvcx	A1	89	Branch on Overflow Clear with Execute	-- none --
bvs [1]	A1	8E	Branch on Overflow Set (long)	-- none --
		0	Branch on Overflow Set (short)	-- none --
bvsr	R1	EE	Branch on Overflow Set Using Register	-- none --
bvsrx	R1	EF	Branch on Overflow Set Using Register with Execute	-- none --
bvsx	A1	8F	Branch on Overflow Set with Execute	-- none --
bx	A1	89	Branch with Execute	-- none --
c	R1,R2	B4	Compare	LT, EQ, GT
cal	R1,D2(R2)	C8	Compute Address Lower Half	-- none --
cal16	R1,D2(R2)	C2	Compute Address Lower Half 16-Bit	-- none --
cas	R1,R2,R3	6	Compute Address Short	-- none --
cau	R1,D2(R2)	D8	Compute Address Upper Half	-- none --
ca16	R1,R2	F3	Compute Address 16-Bit	-- none --
ci [2]	R1,I2	D4	Compare Immediate (long)	LT, EQ, GT
		94	Compare Immediate (short)	LT, EQ, GT
cl	R1,R2	B3	Compare Logical	LT, EQ, GT
cli	R1,I2	D3	Compare Logical Immediate	LT, EQ, GT
clrb	R1,I2	99	Clear Bit (lower half)	LT, EQ, GT
		98	Clear Bit (upper half)	LT, EQ, GT
clrcb [5]	SCR1,I2	95	Clear Bit in the System Control Register	-- any --
clz	R1,R2	F5	Count Leading Zeroes	-- none --
d	R1,R2	B6	Divide Step	C0, OV
dec	R1,I2	93	Decrement	LT, EQ, GT, C0, OV
exts	R1,R2	B1	Extend Sign	LT, EQ, GT
inc	R1,I2	91	Increment	LT, EQ, GT, C0, OV
ior [6]	R1,D2(R2)	CB	Input/Output Read	-- none --
iow [6]	R1,D2(R2)	DB	Input/Output Write	-- none --

Mnemonic	*Operands*	*Op Code*	*Instruction Name*	*CS Bits Changed*
l [2]	R1,D2(R2)	CD	Load (long)	-- none --
		7	Load (short)	-- none --
lc [2]	R1,D2(R2)	CE	Load Character (long)	-- none --
		4	Load Character (short)	-- none --
lh	R1,D2(R2)	DA	Load Half (for immediate data)	-- none --
		EB	Load Half (for data from register)	-- none --
lha [2]	R1,D2(R2)	CA	Load Half Algebraic (long)	-- none --
		5	Load Half Algebraic (short)	-- none --
lis	R1,I2	A4	Load Immediate Short	-- none --
lm	R1,D2(R2)	C9	Load Multiple	-- none --
lps [4]	I1,D2(R2)	D0	Load Program Status	-- none --
lr	R1,R2	6	Load Register	-- none --
m	R1,R2	E6	Multiply Step	C0
mc03 [8]	R1,R2	F9	Move Character Zero From Three	-- none --
mc13 [8]	R1,R2	FA	Move Character One From Three	-- none --
mc23 [8]	R1,R2	FB	Move Character Two From Three	-- none --
mc30 [8]	R1,R2	FD	Move Character Three From Zero	-- none --
mc31 [8]	R1,R2	FE	Move Character Three From One	-- none --
mc32 [8]	R1,R2	FF	Move Character Three From Two	-- none --
mc33 [8]	R1,R2	FC	Move Character Three From Three	-- none --
mfs [5]	SCR1,R2	96	Move From System Control Register	-- none --
mftb	R1,R2	BC	Move From Test Bit	-- none --
mftbi	R1,I2	9D	Move From Test Bit Immediate (lower half)	-- none --
		9C	Move From Test Bit Immediate (upper half)	-- none --
mts [5]	SCR1,I2	B5	Move To System Control Register	-- any --
mttb [8]	R1,R2	BF	Move To Test Bit	-- none --
mttbi [8]	R1,R2	9F	Move To Test Bit Immediate (lower half)	-- none --
		9E	Move To Test Bit Immediate (upper half)	-- none --
n	R1,R2	E5	AND	LT, EQ, GT
nilo	R1,R2,I3	C6	AND Immediate Lower Half Extended Ones	LT, EQ, GT
nilz	R1,R2,I3	C5	AND Immediate Lower Half Extended Zeroes	LT, EQ, GT
niuo	R1,R2,I3	D6	AND Immediate Upper Half Extended Ones	LT, EQ, GT

niuz	R1,R2,I3	D5	AND Immediate Upper Half Extended Zeroes	LT, EQ, GT
nop		6	No Operation	-- none --
o	R1,R2	E3	OR	LT, EQ, GT
oil	R1,R2,I3	C4	OR Immediate Lower	LT, EQ, GT
oiu	R1,R2,I3	C3	OR Immediate Upper	LT, EQ, GT
onec	R1,R2	F4	One's Complement	LT, EQ, GT, C0, OV
s	R1,R2	E2	Subtract	LT, EQ, GT, C0, OV
sar	R1,R2	B0	Shift Algebraic Right	LT, EQ, GT
sari [3]	R1,R2	A0	Shift Algebraic Right Immediate (small)	LT, EQ, GT
		A1	Shift Algebraic Right Immediate (large)	LT, EQ, GT
se	R1,R2	F2	Subtract Extended	LT, EQ, GT, C0, OV
setb	R1,I2	9B	Set Bit (lower half)	LT, EQ, GT
		9A	Set Bit (upper half)	LT, EQ, GT
setcb [5]	SCR1,I2	97	Set Bit in the System Control Register	-- any --
sf	R1,R2	B2	Subtract From	LT, EQ, GT, C0, OV
sfi	R1,R2,I3	D2	Subtract From Immediate	LT, EQ, GT, C0, OV
sis	R1,I2	92	Subtract Immediate Short	LT, EQ, GT, C0, OV
sl	R1,R2	BA	Shift Left	LT, EQ, GT
sli [3]	R1,I2	AA	Shift Left Immediate (small)	LT, EQ, GT
		AB	Shift Left Immediate (large)	LT, EQ, GT
slp	R1,R2	BB	Shift Left Paired	LT, EQ, GT
slpi [3]	R1,I2	AE	Shift Left Paired Immediate (small)	LT, EQ, GT
		AF	Shift Left Paired Immediate (large)	LT, EQ, GT
sr	R1,R2	B8	Shift Right	LT, EQ, GT
sri [3]	R1,I2	A8	Shift Right Immediate (small)	LT, EQ, GT
		A9	Shift Right Immediate (large)	LT, EQ, GT
srp	R1,R2	B9	Shift Right Paired	LT, EQ, GT
srpi [3]	R1,I2	AC	Shift Right Paired Immediate (small)	LT, EQ, GT
		AD	Shift Right Paired Immediate (large)	LT, EQ, GT
st [2]	R1,D2(R2)	DD	Store (long)	-- none --
		3	Store (short)	-- none --
stc [2]	R1,D2(R2)	DE	Store Character (long)	-- none --
		1	Store Character (short)	-- none --
sth [2]	R1,D2(R2)	DC	Store Half (long)	-- none --
		2	Store Half (short)	-- none --
stm	R1,D2(R2)	D9	Store Multiple	-- none --
svc [7]	I1(R1)	C0	Supervisor Call	-- none --
tgte	R1,R2	BD	Trap If Register Greater Than or Equal	-- none --

Mnemonic	*Operands*	*Op Code*	*Instruction Name*	*CS Bits Changed*
ti	I1,R2,I3	CC	Trap On Condition Immediate	-- none --
tlt	R1,R2	BE	Trap If Register Less Than	-- none --
tsh	R1,D2(R2)	CF	Test And Set Half	-- none --
twoc	R1,R2	E4	Two's Complement	LT, EQ, GT, C0, OV
wait [4]	0,0	F0	Wait	-- none --
x	R1,R2	E7	Exclusive OR	LT, EQ, GT
xil	R1,R2,I3	C7	Exclusive OR Immediate Lower Half	LT, EQ, GT
xiu	R1,R2,I3	D7	Exclusive OR Immediate Upper Half	LT, EQ, GT

[1] For branch instructions, "long" means that the instruction contains immediate data between 9 and 20 bits long. "Short" means that the instruction contains immediate data ≤ 8 bits long.

[2] For these storage instructions and the **ci** instruction, "long" means that the instruction contains a displacement between 1 and 16 bits long. "Short" means that the instruction contains a displacement ≤ 4 bits long.

[3] For these shift instructions, "large" means that the instruction contains immediate data with a value between decimal 16 and 31. "Small" means that the instruction contains immediate data with a value between decimal 0 and 15.

[4] You must not use these privileged instructions in non-privileged state.

[5] With certain operands, these are privileged instructions. With other operands, these are non-privileged instructions.

[6] These are non-privileged instructions, but using them can cause conflicts with your operating system. See *Hardware Technical Reference* to learn how to use these instructions.

[7] This instruction is non-privileged, but it acts like a privileged instruction, because it causes a trap to the operating system. See *Virtual Resource Manager Technical Reference* to learn how to use this instruction.

[8] These instructions do not change the Condition Status unless data is moved into the test bit.

Instructions, Indexed by Op Code

Op codes are represented with hexadecimal values. Where parts of instruction names are in parentheses, it means that the assembler language mnemonic has two corresponding op codes. In this case, the assembler examines the instruction's immediate data, then automatically chooses the correct op code.

Machine Op Code	*Assembler Language Mnemonic*	*Format*	*Instruction Name*
0	b	JI	Branch (short)
	bb	JI	Branch on Condition Bit Immediate (short)
	bcc	JI	Branch on Carry Bit Clear (short)
	bcs	JI	Branch on Carry Bit Set (short)
	beq	JI	Branch on Equal (short)
	bge	JI	Branch on Greater Than or Equal (short)
	bgt	JI	Branch on Greater Than (short)
	ble	JI	Branch on Less Than or Equal (short)
	blt	JI	Branch on Less Than (short)
	bnb	JI	Branch on Not Condition Bit Immediate (short)
	bne	JI	Branch on Not Equal (short)
	btc	JI	Branch on Test Bit Clear (short)
	bts	JI	Branch on Test Bit Set (short)
	bvc	JI	Branch on Overflow Clear (short)
	bvs	JI	Branch on Overflow Set (short)
1	stc	D-short	Store Character (short)
2	sth	D-short	Store Half (short)
3	st	D-short	Store (short)
4	lc	D-short	Load Character (short)
5	lha	D-short	Load Half Algebraic (short)
6	cas	X	Compute Address Short
	lr	X	Load Register
	nop	X	No Operation
7	l	D-short	Load (short)
80 thru 87	--none--		--reserved--

Machine Op Code	*Assembler Language Mnemonic*	*Format*	*Instruction Name*
88	b	BI	Branch (long)
	bcc	BI	Branch on Carry Bit Clear (long)
	bge	BI	Branch on Greater Than or Equal (long)
	ble	BI	Branch on Less Than or Equal (long)
	bnb	BI	Branch on Not Condition Bit Immediate (long)
	bne	BI	Branch on Not Equal (long)
	btc	BI	Branch on Test Bit Clear (long)
	bvc	BI	Branch on Overflow Clear (long)
89	bx	BI	Branch with Execute
	bccx	BI	Branch on Carry Bit Clear with Execute
	bgex	BI	Branch on Greater than or Equal with Execute
	blex	BI	Branch on Less Than or Equal with Execute
	bnbx	BI	Branch on Not Condition Bit Immediate with Execute
	bnex	BI	Branch on Not Equal with Execute
	btcx	BI	Branch on Test Bit Clear with Execute
	bvcx	BI	Branch on Overflow Clear with Execute
8A	bala	BA	Branch and Link Absolute
8B	balax	BA	Branch and Link Absolute with Execute
8C	bali	BI	Branch and Link Immediate
8D	balix	BI	Branch and Link Immediate with Execute
8E	bb	BI	Branch on Condition Bit Immediate (long)
	bcs	BI	Branch on Carry Bit Set (long)
	beq	BI	Branch on Equal (long)
	bgt	BI	Branch on Greater Than (long)
	blt	BI	Branch on Less Than (long)
	bts	BI	Branch on Test Bit Set (long)
	bvs	BI	Branch on Overflow Set (long)
	nop	BI	No Operation (long)
8F	bbx	BI	Branch on Condition Bit Immediate with Execute
	bcsx	BI	Branch on Carry Bit Set with Execute
	beqx	BI	Branch on Equal with Execute
	bgtx	BI	Branch on Greater Than with Execute
	bltx	BI	Branch on Less Than with Execute
	btsx	BI	Branch on Test Bit Set with Execute
	bvsx	BI	Branch on Overflow Set with Execute
	nopx	BI	No Operation with Execute
90	ais	R	Add Immediate Short
91	inc	R	Increment

Machine Op Code	*Assembler Language Mnemonic*	*Format*	*Instruction Name*
92	sis	R	Subtract Immediate Short
93	dec	R	Decrement
94	ci	R	Compare Immediate (short)
95	clrcb	R	Clear Bit in the System Control Register
96	mfs	R	Move from System Control Register
97	setcb	R	Set Bit in the System Control Register
98	clrb	R	Clear Bit (upper half)
99	clrb	R	Clear Bit (lower half)
9A	setb	R	Set Bit (upper half)
9B	setb	R	Set Bit (lower half)
9C	mftbi	R	Move From Test Bit Immediate (upper half)
9D	mftbi	R	Move From Test Bit Immediate (lower half)
9E	mttbi	R	Move To Test Bit Immediate (upper half)
9F	mttbi	R	Move To Test Bit Immediate (lower half)
A0	sari	R	Shift Algebraic Right Immediate (small)
A1	sari	R	Shift Algebraic Right Immediate (large)
A2	--none--		-- reserved --
A3	--none--		-- reserved --
A4	lis	R	Load Immediate Short
A5	--none--		-- reserved --
A6	--none--		-- reserved --
A7	--none--		-- reserved --
A8	sri	R	Shift Right Immediate (small)
A9	sri	R	Shift Right Immediate (large)
AA	sli	R	Shift Left Immediate (small)
AB	sli	R	Shift Left Immediate (large)
AC	srpi	R	Shift Right Paired Immediate (small)
AD	srpi	R	Shift Right Paired Immediate (large)
AE	slpi	R	Shift Left Paired Immediate (small)
AF	slpi	R	Shift Left Paired Immediate (large)
B0	sar	R	Shift Algebraic Right
B1	exts	R	Extend Sign
B2	sf	R	Subtract From
B3	cl	R	Compare Logical
B4	c	R	Compare
B5	mts	R	Move to System Control Register
B6	d	R	Divide Step
B7	--none--		-- reserved --

Machine Op Code	*Assembler Language Mnemonic*	*Format*	*Instruction Name*
B8	sr	R	Shift Right
B9	srp	R	Shift Right Paired
BA	sl	R	Shift Left
BB	slp	R	Shift Left Paired
BC	mftb	R	Move From Test Bit
BD	tgte	R	Trap If Register Greater Than or Equal
BE	tlt	R	Trap If Register Less Than
BF	mttb	R	Move To Test Bit
C0	svc	D	Supervisor Call
C1	ai	D	Add Immediate
C2	cal16	D	Compute Address Lower Half 16-Bit
C3	oiu	D	OR Immediate Upper
C4	oil	D	OR Immediate Lower
C5	nilz	D	AND Immediate Lower Half Extended Zeroes
C6	nilo	D	AND Immediate Lower Half Extended Ones
C7	xil	D	Exclusive OR Immediate Lower Half
C8	cal	D	Compute Address Lower Half
C9	lm	D	Load Multiple
CA	lha	D	Load Half Algebraic (long)
CB	ior	D	Input/Output Read
CC	ti	D	Trap On Condition Immediate
CD	l	D	Load (long)
CE	lc	D	Load Character (long)
CF	tsh	D	Test And Set Half
D0	lps	D	Load Program Status
D1	aei	D	Add Extended Immediate
D2	sfi	D	Subtract From Immediate
D3	cli	D	Compare Logical Immediate
D4	ci	D	Compare Immediate (long)
D5	niuz	D	AND Immediate Upper Half Extended Zeroes
D6	niuo	D	AND Immediate Upper Half Extended Ones
D7	xiu	D	Exclusive OR Immediate Upper Half
D8	cau	D	Compute Address Upper Half
D9	stm	D	Store Multiple
DA	lh	D	Load Half (for immediate data)
DB	iow	D	Input/Output Write
DC	sth	D	Store Half (long)
DD	st	D	Store (long)

Machine Op Code	*Assembler Language Mnemonic*	*Format*	*Instruction Name*
DE	stc	D	Store Character (long)
DF	--none--		-- reserved --
E0	abs	R	Absolute
E1	a	R	Add
E2	s	R	Subtract
E3	o	R	OR
E4	twoc	R	Two's Complement
E5	n	R	AND
E6	m	R	Multiply Step
E7	x	R	Exclusive OR
E8	bccr	R	Branch on Carry Bit Clear Using Register
	bger	R	Branch on Greater Than or Equal Using Register
	bler	R	Branch on Less Than or Equal Using Register
	bnbr	R	Branch on Not Condition Bit
	bner	R	Branch on Not Equal Using Register
	br	R	Branch Using Register
	btcr	R	Branch on Test Bit Clear Using Register
	bvcr	R	Branch on Overflow Clear Using Register
E9	bccrx	R	Branch on Carry Bit Clear Using Register with Execute
	bgerx	R	Branch on Greater than or Equal Using Register with Execute
	blerx	R	Branch on Less Than or Equal Using Register with Execute
	bnbrx	R	Branch on Not Condition Bit with Execute
	bnerx	R	Branch on Not Equal Using Register with Execute
	brx	R	Branch Using Register with Execute
	btcrx	R	Branch on Test Bit Clear Using Register with Execute
	bvcrx	R	Branch on Overflow Clear Using Register with Execute
EA	--none--		-- reserved --
EB	lh	R	Load Half (for data from register)
EC	balr	R	Branch and Link Using Register
ED	balrx	R	Branch and Link Using Register with Execute

EE	bbr	R	Branch on Condition Bit
	bcsr	R	Branch on Carry Bit Set Using Register
	beqr	R	Branch on Equal Using Register
	bgtr	R	Branch on Greater Than Using Register
	bltr	R	Branch on Less Than Using Register
	btsr	R	Branch on Test Bit Set Using Register
	bvsr	R	Branch on Overflow Set Using Register
	nopr	R	No Operation Using Register
EF	bbrx	R	Branch on Condition Bit with Execute
	bcsrx	R	Branch on Carry Bit Set Using Register with Execute
	beqrx	R	Branch on Equal Using Register with Execute
	bgtrx	R	Branch on Greater Than Using Register with Execute
	bltrx	R	Branch on Less Than Using Register with Execute
	btsrx	R	Branch on Test Bit Set Using Register with Execute
	bvsrx	R	Branch on Overflow Set Using Register with Execute
	noprx	R	No Operation Using Register with Execute
F0	wait	R	Wait
F1	ae	R	Add Extended
F2	se	R	Subtract Extended
F3	ca16	R	Compute Address 16-Bit
F4	onec	R	One's Complement
F5	clz	R	Count Leading Zeroes
F6	--none--		-- reserved --
F7	--none--		-- reserved --
F8	--none--		-- reserved --
F9	mc03	R	Move Character Zero From Three
FA	mc13	R	Move Character One From Three
FB	mc23	R	Move Character Two From Three
FC	mc33	R	Move Character Three From Three
FD	mc30	R	Move Character Three From Zero
FE	mc31	R	Move Character Three From One
FF	mc32	R	Move Character Three From Two

E8	bccr	R	Branch on Carry Bit Clear Using Register
	bger	R	Branch on Greater Than or Equal Using Register
	bler	R	Branch on Less Than or Equal Using Register
	bnbr	R	Branch on Not Condition Bit
	bner	R	Branch on Not Equal Using Register
	br	R	Branch Using Register
	btcr	R	Branch on Test Bit Clear Using Register
	bvcr	R	Branch on Overflow Clear Using Register
E9	bccrx	R	Branch on Carry Bit Clear Using Register with Execute
	bgerx	R	Branch on Greater than or Equal Using Register with Execute
	blerx	R	Branch on Less Than or Equal Using Register with Execute
	bnbrx	R	Branch on Not Condition Bit with Execute
	bnerx	R	Branch on Not Equal Using Register with Execute
	brx	R	Branch Using Register with Execute
	btcrx	R	Branch on Test Bit Clear Using Register with Execute
	bvcrx	R	Branch on Overflow Clear Using Register with Execute
EA	--none--		-- reserved --
EB	lh	R	Load Half (for data from register)
EC	balr	R	Branch and Link Using Register
ED	balrx	R	Branch and Link Using Register with Execute
EE	bbr	R	Branch on Condition Bit
	bcsr	R	Branch on Carry Bit Set Using Register
	beqr	R	Branch on Equal Using Register
	bgtr	R	Branch on Greater Than Using Register
	bltr	R	Branch on Less Than Using Register
	btsr	R	Branch on Test Bit Set Using Register
	bvsr	R	Branch on Overflow Set Using Register
	nopr	R	No Operation Using Register

EF	bbrx	R	Branch on Condition Bit with Execute
	bcsrx	R	Branch on Carry Bit Set Using Register with Execute
	beqrx	R	Branch on Equal Using Register with Execute
	bgtrx	R	Branch on Greater Than Using Register with Execute
	bltrx	R	Branch on Less Than Using Register with Execute
	btsrx	R	Branch on Test Bit Set Using Register with Execute
	bvsrx	R	Branch on Overflow Set Using Register with Execute
	noprx	R	No Operation Using Register with Execute
F0	wait	R	Wait
F1	ae	R	Add Extended
F2	se	R	Subtract Extended
F3	ca16	R	Compute Address 16-Bit
F4	onec	R	One's Complement
F5	clz	R	Count Leading Zeroes
F6	--none--		-- reserved --
F7	--none--		-- reserved --
F8	--none--		-- reserved --
F9	mc03	R	Move Character Zero From Three
FA	mc13	R	Move Character One From Three
FB	mc23	R	Move Character Two From Three
FC	mc33	R	Move Character Three From Three
FD	mc30	R	Move Character Three From Zero
FE	mc31	R	Move Character Three From One
FF	mc32	R	Move Character Three From Two

Appendix B. ASCII Character Codes

First Hexadecimal Digit

Second Hexadecimal Digit	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	NUL	DLE	BLANK (SPACE)	0	@	P	`	p	Ç	É	á	░	└	╨	α	≡
1	SOH	DC1	!	1	A	Q	a	q	ü	æ	í	▒	┴	╤	β	±
2	STX	DC2	"	2	B	R	b	r	é	Æ	ó	▓	┬	╥	Γ	≥
3	ETX	DC3	#	3	C	S	c	s	â	ô	ú	│	├	╙	π	≤
4	EOT	DC4	$	4	D	T	d	t	ä	ö	ñ	┤	─	╘	Σ	⌠
5	ENQ	NAK	%	5	E	U	e	u	à	ò	Ñ	╡	┼	╒	σ	⌡
6	ACK	SYN	&	6	F	V	f	v	å	û	ª	╢	╞	╓	µ	÷
7	BEL	ETB	'	7	G	W	g	w	ç	ù	º	╖	╟	╫	τ	≈
8	BS	CAN	(	8	H	X	h	x	ê	ÿ	¿	╕	╚	╪	Φ	°
9	HT	EM	)	9	I	Y	i	y	ë	Ö	⌐	╣	╔	┘	θ	•
A	LF	SUB	*	:	J	Z	j	z	è	Ü	¬	║	╩	┌	Ω	·
B	VT	ESC	+	;	K	[	k	{	ï	¢	½	╗	╦	█	δ	√
C	FF	FS	,	<	L	\	l	¦	î	£	¼	╝	╠	▄	∞	ⁿ
D	CR	GS	—	=	M	]	m	}	ì	¥	¡	╜	═	▌	ø	²
E	SO	RS	.	>	N	^	n	~	Ä	₧	«	╛	╬	▐	ε	■
F	SI	US	/	?	O	_	o	⌂	Å	ƒ	»	┐	╧	▀	∩	BLANK FF

See the Notes on the next page.

Notes:

1. ASCII characters 00 through 1F are control characters that do not display on the screen. These control characters have the following meaning:

NUL	null	DLE	data link escape
SOH	start of heading	DC1	device control character
STX	start of text	DC2	device control character
ETX	end of text	DC3	device control character
EOT	end of transmission	DC4	device control character
ENQ	enquiry	NAK	negative acknowledge
ACK	acknowledge	SYN	synchronous idle
BEL	bell	ETB	end of transmission block
BS	backspace	CAN	cancel
HT	horizontal tab	EM	end of medium
LF	line feed	SUB	substitute
VT	vertical tab	ESC	escape
FF	form feed	FS	file separator
CR	carriage return	GS	group separator
SO	shift out	RS	record separator
SI	shift in	US	unit separator

2. The ASCII characters in the chart represent a default for the AIX Operating System using a monochrome display. If you are using a color display, you may see different ASCII characters. Applications running in the AIX Operating System may also cause different ASCII characters to display.

Figures

Glossary

032 Microprocessor. The 32-bit processor for the RT PC system.

032 Microprocessor instructions. Specify an operation to be performed by the processor, along with the values or locations of operands, if any exist. Each 032 Microprocessor instruction has a mnemonic corresponding to the numerical op code(s) of machine language instruction(s).

absolute value. The numeric value of a real number regardless of its sign (positive or negative).

address. A number that identifies a location in memory.

addressing. A means of identifying storage locations.

algebraic comparison. When two numbers are compared, their high-order bits are treated as signed bits. Algebraically, hex FFFF0000 is less than hex 00000001, because hex FFFF0000 algebraically represents a negative number. Contrast with *logical comparison.*

allocate. To assign a resource to perform a specific task.

alphabetic. Pertaining to a set of letters a through z.

American National Standard Code for Information Interchange (ASCII). The code developed by ANSI for information interchange among data processing systems, data communications systems, and associated equipment. The ASCII character set consists of control characters and symbolic characters.

a.out. The name of the default output file produced by **as**. *AIX Operating System Technical Reference* shows the format of this file.

application. A particular task, such as inventory control or accounts receivable. Programs written to perform an application usually run above the operating system level, and are written in a high-level programming language.

ASCII. See *American National Standard Code for Information Interchange.*

assembler. A program that translates an assembler language source module to a machine language object module. The assembler converts mnemonic op codes to numeric, machine-readable op codes and converts implicit addresses to displacements and base register numbers.

assembler directives. See *pseudo-operations.*

assembler instructions. See *instruction* and *pseudo-operations.*

assembler sections. See *section.*

base address. The beginning address for resolving symbolic references to locations in storage. An address that is defined, in part, by the contents of a base register at runtime. See **effective address.**

base register. A register whose contents are added to a displacement to form an effective address.

basic addressable unit (BAU). The smallest piece of storage that can be addressed. On the RT PC, a byte is the BAU.

BAU. See *basic addressable unit.*

block. See *program block.*

boundary alignment. The position in main storage of a fixed-length field (such as halfword or doubleword) on an integral boundary for that unit of information. For example, a word boundary is a storage address evenly divisible by four.

branch. An instruction that changes the sequence of instruction execution by putting the address of another instruction into the Instruction Address Register. A branch can occur conditionally or non-conditionally. A conditional branch occurs only when a specified condition is met.

byte. The amount of storage required to represent one character. A byte is 8 bits.

call. To activate a program or procedure at its entry point. Compare with *load.*

character set. A group of characters used for a specific reason; for example, the set of characters the assembler recognizes.

check. (1) An error condition. (2) To look for a condition.

comments field. The last field of an assembler instruction. All comments must be preceded by a pound sign (#).

compiler. A program that translates instructions written in a specific high-level programming language into machine language.

constant. A data item with a value that does not change.

counter. A register or storage location used to accumulate the number of occurrences of an event.

delimiter. A character or sequence of characters that marks the beginning or end of a unit of data.

displacement. A number that can be added to the contents of the base register to calculate an effective address.

editor. A utility that programmers use to enter and modify source code.

effective address. A real storage address that is computed at runtime. The effective address consists of contents of a base register plus a displacement.

exception handler. A set of routines used to detect deadlock conditions or to process abnormal condition processing. This allows the normal execution of processes to be interrupted and resumed.

expression. A representation of a value. For example, variables and constants appearing alone or in combination with operators.

extension. See *sign extension.*

external reference. A reference to a symbol that is defined as a global name in another module. Also, a symbol that is not defined in the module that references it.

external symbol. An external reference that is defined or referred to in a particular module. An ordinary symbol that represents an external reference.

file. Synonymous with *module.*

file name. The name used by a program to identify a file.

file section. In assembler language source code, the smallest group of statements that can be assembled. Pseudo-ops define assembler language file sections for instructions, data, and uninitialized data. The link editor translates assembler file sections to runtime (**a.out**) segments.

forward reference. In a statement, referring to a symbol that has not yet been defined.

function. A block of assembler language statements that do a limited task. A function or functions can optionally be a subset of the instructions contained in a single assembler file section.

general purpose register (GPR). An explicitly addressable register that can be used for a variety of purposes (for example, as an accumulator or a base register). RT PC has 16 32-bit GPRs. See *register.*

global symbol. A symbol defined in one program module, but used without redefinition in other independently assembled program modules.

GPR. See *general purpose register.*

high-order. Most significant; leftmost. For example, bit 0 in a register.

IAR. See *instruction address register.*

immediate data. Data appearing in an instruction itself (as opposed to the symbolic name of the data). The data is immediately available from the instruction and therefore is assembled directly into the instruction without being read from memory.

instruction. Assembler language programs are made up of 032 Microprocessor instructions and pseudo-op instructions. See *pseudo-operations* and *032 Microprocessor instructions.*

instruction address register (IAR). A system control register that contains the address of the next instruction to be executed (that is, the updated instruction address). The IAR (sometimes called a "program counter") is incremented in bytes. Compare with *location counter.*

instruction format. The allocation of bits or characters of a machine instruction to specific classes of instructions.

interrupt. A signal sent by an I/O device to the processor when an error has occurred or when assistance is needed to complete I/O. An interrupt usually suspends execution of the currently executing program.

jump. A short branch. The programmer cannot explicitly issue a jump instruction.

label. The field of an instruction that assigns a symbolic name to the location at which the instruction begins.

link editor. A utility that resolves cross-references between separately assembled object modules, then assigns final addresses to create a single relocatable load module. If a single object module is linked, the linker simply makes it relocatable. (Also called "linker" or "linkage editor.")

linker. See *link editor.*

load. To move data or programs into storage.

loader. A program that moves data or other programs into storage. In AIX Operating System, the loader runs when you type a program name to run the program. See *link editor.*

load module. The output of the linker. A program in a format suitable for loading into main storage and executing.

location counter. A counter in the assembler that denotes the next byte available for code allocation. The location counter assigns storage addresses to program statements. On RT PC, the text and data assembler file sections each have four location counters. Compare with *Instruction Address Register.*

logical comparison. When comparing two numbers, their high-order bit is treated as an unsigned bit. Logically, hex FFFF0000 is greater than hex 00000001. Contrast with *algebraic comparison.*

low-order. Least significant; rightmost. For example, bit 31 in a 32-bit register.

machine instruction. A series of binary numbers directing the operation of a processor. The assembler converts 032 Microprocessor instruction mnemonics and operands to machine instructions.

machine language. A language consisting of machine instructions that can be used directly by a computer without intermediate processing.

macro. A single instruction representing a series of instructions. RT PC 032 Microprocessor assembler language does not have macros.

main storage. The part of the processing unit where programs are run.

mask. A pattern of characters that controls the keeping, deleting, or testing of portions of another pattern of characters.

mnemonic. The field of an instruction that contains the acronym or abbreviation for a machine instruction or pseudo-op. For machine instructions, using mnemonics frees the programmer from having to remember the machine's numeric op codes.

module. For assembler language source code, one or more assembler file sections. Modules are subroutines, calling programs, and data areas that are assembled separately, then linked to make a complete program. See *load module* and *object module.*

non-volatile. A register whose value on subroutine entry must be preserved on subroutine exit. Constrast with *volatile.*

object code. See *object module.*

object module. A set of instructions in machine language. The object module is produced by an assembler from a subroutine or source module and can be input to the link editor. See *module.*

op code. See *operation code.*

operand. An instruction field that represents data (or the location of data) to be manipulated or operated upon. Not all instructions require an operand field.

operating system state. The state in which the AIX Operating System kernel runs. A virtual machine protection state which occurs in 032 Microprocessor unprivileged state. (See *Virtual Resource Manager Technical Reference.*)

operation. A specific action (such as move, add, multiply, load) that the computer performs when requested.

operation code. A numeric code indicating to the processor which operation should be performed.

overflow condition. A condition that occurs when a portion of the result of an operation exceeds the capacity of the intended unit of storage.

pipelined. A processor that can process two or more instructions at the same time. A pipelined processor simultaneously fetches one instruction, fetches a second instruction's operands, and performs an operation for a third instruction. This allows instructions to run faster; for example, the subject of a branch-with-execute instruction runs at the same time that the pipeline is loaded with the address of the target.

privileged instruction. Can be executed only when 032 Microprocessor is in the privileged state. Application programs do not usually contain privileged instructions.

problem state. A virtual machine protection state which occurs in 032 Microprocessor unprivileged state. See *Virtual Resource Manager Technical Reference.*

program block. Used to construct symbol table information for block-structured languages, such as C, which pass source symbol information through a compiler and into an assembler.

program counter. See *instruction address register.*

pseudo-operations. Functions that the assembler performs at assembly time, not at runtime. Instructions to the assembler itself, such as assembler file section assignment and byte alignment. (Also called "pseudo-ops," "assembler directives," and "assembler operators.")

register. In a computer, a storage area capable of storing a specified amount of data such as a bit or an address. On RT PC, each register is at most 32 bits long. See *general purpose register* and *system control register.*

register pair. The pair of a general purpose register is the binary value of the register with the low-order bit inverted. For example, register 5 (binary 0101) and register 4 (binary 0100) are pairs.

relative address. An address specified in relation to the contents of the Instruction Address Register or to a symbol. When a program is relocated, the addresses themselves will change, but the specification of relative addresses remains the same.

relative addressing. A means of addressing instructions and data areas by designating their locations to the Instruction Address Register or to some symbol.

relocatable. A value, expression, or address is relocatable if it does not have to be changed when the program is relocated.

relocation. Changing address constants so that a program can be executed from an area of memory different from the area assigned during assembly.

run-time environment. A collection of subroutines that provide commonly used functions for system components.

section. In assembler language source code, a group of statements demarcated by **.text, .data,** or **.lcomm** pseudo-ops. An assembler language section maps to a segment in the **a.out** file at runtime.

segment. A contiguous area of virtual storage allocated to a job or system task. On RT PC, the executable **a.out** file contains segments which correspond to assembler language sections.

sign extension. At runtime, duplicating the high-order bit of a number throughout any vacant high-order positions that exist when the number is expanded to 32 bits. For example, hex 8000 sign-extended is hex FFFF8000. Sign extending immediate data and shifting it left one bit has the same effect as multiplying the immediate data by 2. Sign extension thereby lets immediate data be evaluated in terms of halfwords, not bytes (since the instructions must be on halfword boundaries).

source module. The statements that form input to the assembler.

special character. A character other than an alphabetic or numeric character. For example, *, +, and % are special characters.

stack. An area in storage that stores temporary register information and return addresses of subroutines. 032 Microprocessor does not have hardware support for a stack.

stack pointer. A register providing the current location of the stack.

standard error. For certain commands, a default location to which error messages are sent, usually the terminal.

statement. An instruction in a program or procedure.

store. To place information into memory where it is available for retrieval and updating.

supervisor. The part of RT PC that coordinates the use of resources and maintains the flow of processing unit operations.

supervisor call (svc). An instruction that interrupts the program being executed and passes control to the supervisor so it can perform a specific service indicated by the instruction.

supervisor state. A virtual machine protection state corresponding to 032 Microprocessor privileged state. See *Virtual Resource Manager Technical Reference.* (Contrast with *problem state.*)

svc. See *supervisor call.*

symbol table. Control information, associated with an object or load module, that is produced by the assembler and identifies the external symbols in the module.

symbolic debugger (sdb). An AIX Operating System command that debugs programs that conform to the format of the **a.out** object file.

system control register (SCR). One of 16 hardware registers used to control the state of the processor. The IAR, MQ, and CS are all SCRs.

term. The smallest part of an expression that can be assigned a value.

trap. An unprogrammed, hardware-initiated jump to a specific address. Occurs as a result of an error or certain other conditions.

two's complement. Representation of negative binary numbers. Formed by subtracting each digit of the number from zero, then adding one to the result.

unprivileged instruction. Can be executed when the 032 Microprocessor is in privileged or unprivileged state. The unprivileged instructions consist of ordinary loads, stores, adds, and so forth.

updated instruction address. The value in the IAR that refers to the address of the next instruction to be executed, not to the address of the currently executing instruction.

virtual machine. A functional simulation of a computer and its associated devices. AIX Operating System controls the concurrent execution of many virtual machines, including execution of software.

virtual machine interface (VMI). A software interface between hardware and an operating system. The VMI shields operating system software from hardware changes and low-level interfaces, and provides for concurrent execution of multiple virtual machines.

virtual resource manager (VRM). A set of programs that manage the hardware resources (main storage, disk storage, display stations, and printers) of the system so that these resources can be used independently of each other.

virtual storage. Addressable space that appears to be real storage. From virtual storage, instructions and data are mapped into real storage locations.

VMI. See *virtual machine interface.*

volatile. A register whose value on subroutine entry does not need to be preserved on subroutine exit. Contrast with *non-volatile.*

VRM. See *virtual resource manager.*

word. A contiguous series of 32 bits (four bytes) in storage, addressable as a unit. The address of the first byte of a word is evenly divisible by four.

wraparound. (1) The continuation of addresses from the highest allowable address to the lowest; the maximum address is followed by address 0. (2) In an arithmetic operation, the continuation of values from the highest allowable value to the lowest; the maximum value is followed by a value of 0. Whenever this type of wraparound occurs, the overflow bit in the condition status register is set to 1.

Index

IBM RT PC

Reader's Comment Form

IBM RT PC Assembler Language Reference

SV21-8011-0

Your comments assist us in improving our products. IBM may use and distribute any of the information you supply in any way it believes appropriate without incurring any obligation whatever. You may, of course, continue to use the information you supply.

For prompt resolution to questions regarding set up, operation, program support, and new program literature, contact the authorized IBM RT PC dealer in your area.

Comments:

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 40 ARMONK, NEW YORK

POSTAGE WILL BE PAID BY ADDRESSEE

International Business Machines Corporation
Department 997, Building 998
11400 Burnet Rd.
Austin, Texas 78758

Fold and tape

Fold and tape

Cut or Fold Along Line

Tape

Please Do Not Staple

Tape

IBM RT PC

Reader's Comment Form

IBM RT PC Assembler Language Reference

SV21-8011-0

Your comments assist us in improving our products. IBM may use and distribute any of the information you supply in any way it believes appropriate without incurring any obligation whatever. You may, of course, continue to use the information you supply.

For prompt resolution to questions regarding set up, operation, program support, and new program literature, contact the authorized IBM RT PC dealer in your area.

Comments:

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 40 ARMONK, NEW YORK

POSTAGE WILL BE PAID BY ADDRESSEE

International Business Machines Corporation
Department 997, Building 998
11400 Burnet Rd.
Austin, Texas 78758

Fold and tape

Fold and tape

Cut or Fold Along Line

Tape

Please Do Not Staple

Tape

International Business
Machines Corporation
Department 997, Building 998
11400 Burnet Rd.
Austin, Texas 78758

Printed in the
United States of America

59X7994