AIX Version 3 for
RISC System/6000™

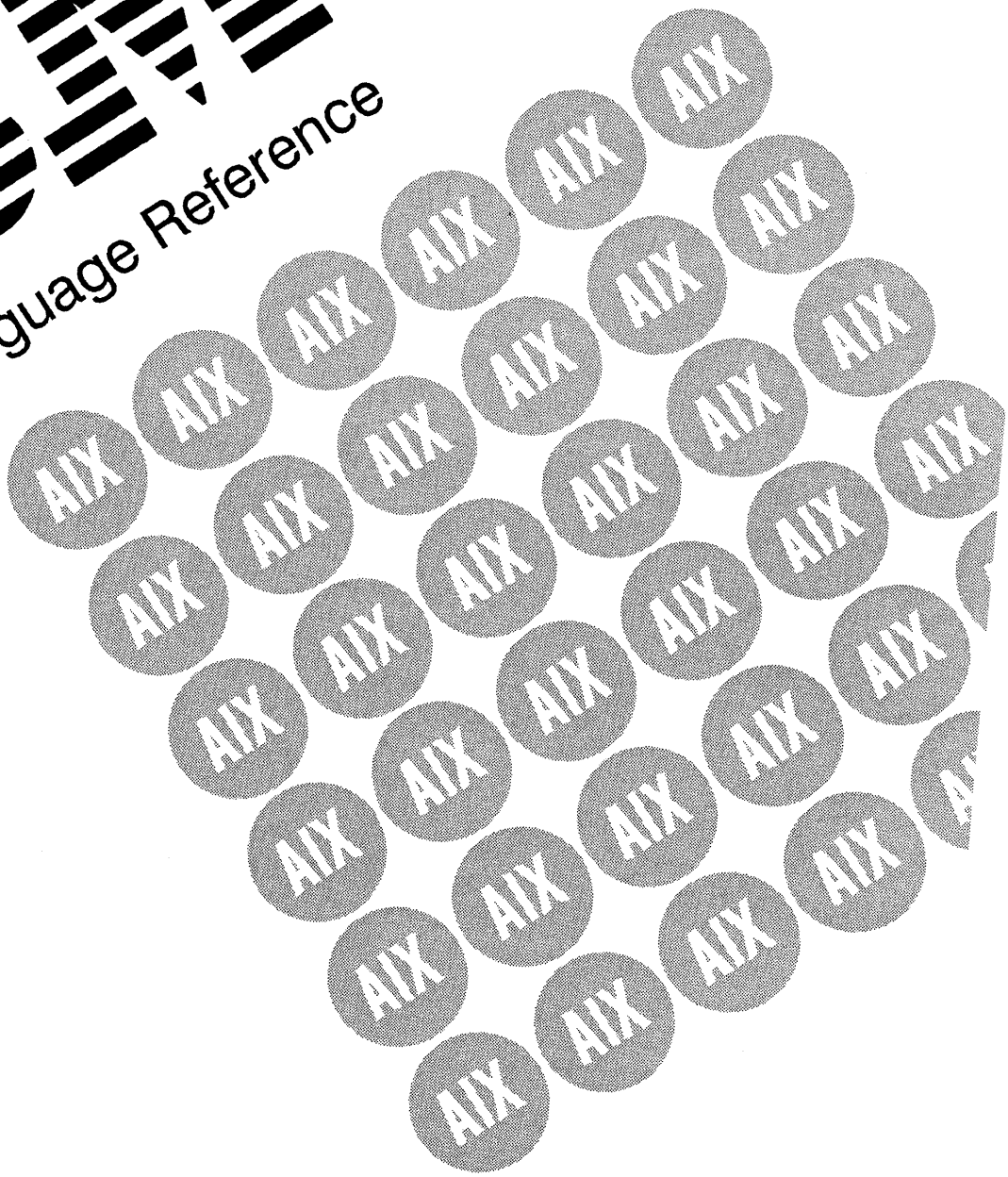
Assembler Language Reference

SC2

IBM

AIX Version 3 for
RISC System/6000™

Assembler Language Reference



First Edition (March 1990)

This edition of the *Assembler Language Reference for IBM AIX Version 3 for RISC System/6000* applies to Version 3 of IBM AIX RISC System/6000 Licensed Program and to all subsequent releases of these products until otherwise indicated in new releases or technical newsletters.

The following paragraph does not apply to the United Kingdom or any country where such provisions are inconsistent with local law: INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS MANUAL "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions; therefore, this statement may not apply to you.

This publication could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time.

It is possible that this publication may contain reference to, or information about, IBM products (machines and programs), programming, or services that are not announced in your country. Such references or information must not be construed to mean that IBM intends to announce such IBM products, programming, or services in your country. Any reference to an IBM licensed program in this publication is not intended to state or imply that you can use only IBM's licensed program. You can use any functionally equivalent program instead.

Requests for copies of this publication and for technical information about IBM products should be made to your IBM Authorized Dealer or your IBM Marketing Representative.

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 78758–3493. IBM may use or distribute whatever information you supply in any way it believes appropriate without incurring any obligation to you.

Trademarks and Acknowledgements

The following trademarks and acknowledgements apply to this book:

AIX is a trademark of International Business Machines Corporation.

IBM is a registered trademark of International Business Machines Corporation.

RISC System/6000 is a trademark of International Business Machines Corporation.

UNIX was developed and licensed by AT&T and is a registered trademark of AT&T Corporation.

About This Book

This book provides information on application programming interfaces to the Advanced Interactive Executive Operating System (referred to in this text as AIX) for use on the RISC System/6000.

Who Should Use This Book

This book is intended for experienced assembler language programmers. To use this book effectively, you should be familiar with AIX or UNIX System V commands, assembler instructions and pseudo–ops, and processor register usage.

How to Use This Book

Overview of Contents

This book contains the following sections consisting of processor information, syntax and semantics, addressing information, the instruction set, information on running a program, and pseudo–ops.

- Overview of Processing and Storage on the RISC System/6000 Microprocessor
- Syntax and Semantics Overview
- Addressing Overview
- Instruction Set in alphabetical order
- Assembling, Linking, and Running a Program Overview
- Pseudo–ops Overview and alphabetical listing of Pseudo–ops

Highlighting

The following highlighting conventions are used in this book:

Bold	Identifies instructions, pseudo–ops, commands, and other items whose names are predefined by the system.
Italics	Identifies parameters whose actual names or values are to be supplied by the user.
`Monospace`	Identifies examples of specific data values, examples of text similar to what you might see displayed, examples of portions of program code similar to what you might write as a programmer, messages from the system, or information you should actually type.

What is Not in This Book

This book does not teach readers how to program or operate their IBM RISC System/6000. 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 limited information on the **as**, **ld**, and **cc** commands.

- Error messages generated by the **as** command. These messages are shown in *Task Index and Glossary for IBM RISC System/6000.*
- Any hardware features. This book does give brief explanations about some processor registers and their use.
- Details about privileged instructions.

Related Publications

The following books contain information about or related to assembler language programming:

- *IBM RISC System/6000* in *POWERstation and POWERserver Hardware Technical Reference — General Information*, SA23–2643.
- *AIX Files Reference for IBM RISC System/6000*, SC23–2200.
- *AIX Commands Reference for IBM RISC System/6000*, SC23–2199.
- *Task Index and Glossary for IBM RISC System/6000*, SC23–2201.

Ordering Additional Copies of This Book

To order additional copies of this book, use Order Number SC23–2197.

Contents

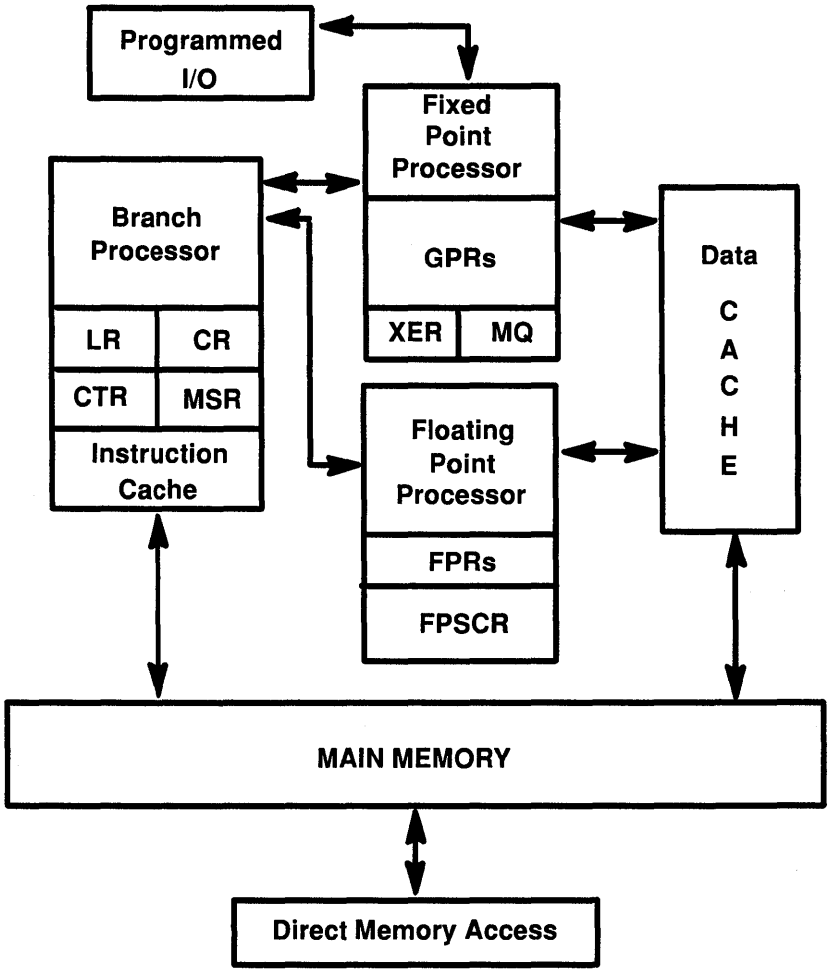
Chapter 1. Processing and Storage

Overview of Processing and Storage on the RISC System/6000 Microprocessor

The characteristics of the RISC System/6000 Microprocessor's processor and storage influence its assembler language. The capabilities of the processor and the nature of available storage determine what the assembler can do.

This chapter gives an overview of the RISC System/6000 Microprocessor and tells you how data is stored both in main memory and in registers. This information will give you some of the conceptual background necessary to understanding the function of the RISC System/6000 Microprocessor's instruction set and pseudo-ops.

The processor unit contains the sequencing and processing controls for instruction fetch, instruction execution and interrupt action. The following figure shows the logical partitioning of the RISC System/6000 Microprocessor.



The processing unit is a word oriented fixed point processor functioning in tandem with a doubleword oriented Floating Point processor. The microprocessor uses 32-bit word-aligned instructions and provides for byte, halfword, word, and doubleword operand fetches and stores between storage and a set of 32 General Purpose Registers, and between storage and a set of 32 Floating Point Registers.

Branch Processor Overview

The Branch Processor has four 32-bit registers.

- The Condition Register
- The Link Register
- The Count Register
- The Machine State Register

Branch, Supervisor Linkage, Trap, Return From Interrupt, and Condition Register instructions effect the various registers of the Branch Processor.

Understanding Branch Instructions

Use branch instructions to change the sequence of instruction execution.

Since all branch instructions are on word boundaries, the processor performing the branch ignores bits 30 and 31 of the generated branch target address. All branch instructions can be used in unprivileged state.

A branch instruction computes the target address in one of four ways:

- The target address is the sum of a constant and the address of the branch instruction itself.
- The target address is the absolute address given as argument to the instruction.
- The target address is the address found in the Link Register.
- The target address is the address found in the Count Register.

Using the first two of these methods, the target address can be computed sufficiently ahead of the branch instructions to prefetch instructions along the target path.

Using the third and fourth methods, prefetching instructions along the branch path is also possible provided the Link Register or the Count Register is loaded sufficiently ahead of the branch instruction.

In the case of conditional branch instructions, instruction prefetching is done on each path of the branch.

In the various target forms, branches generally either branch only, branch and provide a return address, branch conditionally, or branch conditionally and provide a return address. If the particular branch instruction has the **l** in the syntax form, then it sets the link bit to 1 and the Link Register stores the return address from an invoked subroutine. The return address is the address of the instruction immediately following the branch instruction.

b (Branch) Instruction

ba (Branch Absolute) Instruction

bb (Branch on Condition Register Bit) Instruction

bc (Branch Conditional) Instruction

bca (Branch Conditional Absolute) Instruction

bcc (Branch Conditional to Count Register) Instruction

bcr (Branch Conditional Register) Instruction

Conditional branch instructions require the specification of branch option bits and condition type and may use an optional link flag.

Extended Mnemonics

These extended instructions are based on branch instructions and facilitate setting specified bits. The branch condition is specified by a Branch Code which replaces the **XX** in each Extended Mnemonic instruction.

bXX, bXXl (Branch on Condition Extended) Instruction

bXXa, bXXla (Branch on Condition Extended Absolute) Instruction

bXXc, bXXcl (Branch Count Register on Condition) Instruction

bXXr, bXXrl (Branch Register on Condition) Instruction

bbta, bbfa, bbtla, bbfla (Branch on Condition Register Bit) Instruction

bbtc, bbfc, bbtcl, bbfcl (Branch on Condition Register Bit) Instruction

bbtr, bbfr, bbfrl, bbtrl (Branch on Condition Register Bit) Instruction

bctr (Branch to Count Register) Instruction

bdz, bdn, bdzl, bdnl (Branch and Decrement CTR) Instruction

bdzXX, bdnXX (Branch and Decrement CTR on Condition) Instruction

bdza, bdna, bdzla, bdnla (Branch Absolute and Decrement CTR) Instruction

bdzr, bdnr, bdzrl, bdnrl (Branch Register and Decrement CTR) Instruction

Branch Code	Meaning
lt	less than*
gt	greater than*
eq	equal to*
so	summary overflow*
ge	greater than or equal to*
le	less than or equal to*
ne	not equal to*
ns	not summary overflow*
nl	not less than
ng	not greater than
z	zero
nz	not zero

Note: The **bdzXX** and **bdnXX** instructions use only the first eight Branch Codes (marked by *) in place of **XX**.

Understanding Supervisor Linkage Instructions

There are two Branch Processor instructions for system control.

svc (Supervisor Call) Instruction

svca (Supervisor Call Absolute) Instruction

Understanding Trap Instructions

You can use the trap instructions to test for a specified set of conditions during the execution of your program. You can define traps for events that should not occur during program execution, such as an index out or range, or the use of an invalid character. If any of the defined trap conditions are met, a Program Interrupt occurs. If the tested conditions are not met, instruction execution continues normally.

These instructions compare the contents of a General Purpose Register with a 32-bit value. This comparison results in five test conditions. These are ANDed with five condition bits provided in the instruction TO field. If the result is not zero, then a Program interrupt occurs.

The five test conditions are:

TO bit	ANDed with Condition
6	Compares Less Than
7	Compares Greater Than
8	Compares Equal
9	Compares Logically Less Than
10	Compares Logically Greater Than

The available Trap Instructions are:

t (Trap) Instruction

ti (Trap Immediate) Instruction

Understanding Condition Register Instructions

Condition Register Field Instructions

The following instruction copies one Condition Register field to another.

mcrf (Move Condition Register Field) Instruction

Condition Register Logical Instructions

The following instructions perform logical operations with the Condition Register fields.

crand (Condition Register AND) Instruction

crandc (Condition Register AND with Complement) Instruction

creqv (Condition Register Equivalent) Instruction

crnand (Condition Register NAND) Instruction

crnor (Condition Register NOR) Instruction

cror (Condition Register OR) Instruction

crorc (Condition Register OR with Complement) Instruction

crxor (Condition Register XOR) Instruction

Related Information

Fixed Point Processor Overview on page 1–6, Floating Point Processor Overview on page 1–12.

See the following articles in *POWERstation and POWERserver Hardware Technical Reference — General Information*: Condition Register, Link Register, Count Register, Machine State Register.

Fixed Point Processor Overview

The Fixed Point Processor uses a set of 32-bit Registers that includes:

- Thirty–two 32-bit General Purpose Registers
- One 32-bit Fixed Point Exception Register
- One 32-bit Multiply Quotient Register

Understanding General Purpose Registers

The thiry–two 32-bit General Purpose Registers constitute the principal internal storage mechanism in the Fixed Point Processor.

Each General Purpose Register is 32 bits wide.

Understanding Fixed Point Load Instructions

The Fixed Point Load Instructions move information from a location in memory into one of the General Purpose Registers.

The Load instructions compute the effective address when moving data. If the storage access does not cause an Alignment Interrupt or a Data Storage Interrupt, the byte, halfword, or word in storage addressed by the effective address is loaded into a target General Purpose Register.

The following Load instructions are available.

l (Load) Instruction

lbrx (Load Byte Reverse Indexed) Instruction

lbz (Load Byte and Zero) Instruction

lbzx (Load Byte and Zero Indexed) Instruction

lha (Load Half Algebraic) Instruction

lhax (Load Half Algebraic Indexed) Instruction

lhbrx (Load Half Byte Reverse Indexed) Instruction

lhz (Load Half and Zero) Instruction

lhzx (Load Half and Zero Indexed) Instruction

lil (Load Immediate Lower) Instruction

liu (Load Immediate Upper) Instruction

lm (Load Multiple) Instruction

lx (Load Indexed) Instruction

Understanding Fixed Point Load with Update Instructions

The Fixed Point Load with Update Instructions move information from a location in memory into one of the General Purpose Registers.

The Load With Update instructions compute an effective address when moving data. This address can replace the contents of the Base General Purpose Register. If the storage

access does not cause an Alignment Interrupt or Data Storage Interrupt, the instruction then copies the byte, halfword, or word contents of the specified location in memory into a second target General Purpose Register.

There are four conditions under which a newly calculated effective address is *not* saved.

- The General Purpose Register to be updated is the same as the target General Purpose Register. Under this circumstance the updated Register contains data loaded from memory.
- The General Purpose Register to be updated is GPR 0.
- The storage access causes an Alignment Interrupt.
- The storage access causes a Data Storage Interrupt.

The following Load with Update Instruction are available:

lbzu (Load Byte And Zero With Update) Instruction

lbzux (Load Byte And Zero With Update Indexed) Instruction

lhau (Load Half Algebraic With Update) Instruction

lhaux (Load Half Algebraic With Update Indexed) Instruction

lhzu (Load Half And Zero With Update) Instruction

lhzux (Load Half And Zero With Update Indexed) Instruction

lu (Load With Update) Instruction

lux (Load With Update Indexed) Instruction

Understanding Fixed Point Store Instructions

If the storage access does not cause an Alignment Interrupt or a Data Storage Interrupt, the contents of a source General Purpose Register are stored into the byte, halfword, or word in storage addressed by the effective address.

st (Store) Instruction

stb (Store Byte) Instruction

stbrx (Store Byte Reverse Indexed) Instruction

stbx (Store Byte Indexed) Instruction

sth (Store Half) Instruction

sthbrx (Store Half Byte Reverse Indexed) Instruction

sthx (Store Half Indexed) Instruction

stm (Store Multiple) Instruction

stx (Store Indexed) Instruction

Understanding Fixed Point Store with Update Instructions

If the storage access does not cause an Alignment Interrupt or a Data Storage Interrupt, the contents of a source General Purpose Register are stored into the byte, halfword, or word in storage addressed by the effective address . If the General Purpose Register does not

contain the address to be updated and is not General Purpose Register 0 and no interrupt occurs, then the effective address is placed into the Base General Purpose Register.

stbu (Store Byte With Update) Instruction

stbux (Store Byte With Update Indexed) Instruction

sthu (Store Half With Update) Instruction

sthux (Store Half with Update Indexed) Instruction

stu (Store With Update) Instruction

stux (Store With Update Indexed) Instruction

Understanding Fixed Point String Instructions

The Fixed Point String Instructions allow the movement of data from storage to registers or from registers to storage without concern for alignment. These instructions can be used for a short move between arbitrary storage locations or to initiate a long move between unaligned storage fields. Load String Indexed and Store String Indexed Instructions of zero length do not alter the target register.

lscbx (Load String And Compare Byte Indexed) Instruction

lsi (Load String Immediate) Instruction

lsx (Load String Indexed) Instruction

stsi (Store String Immediate) Instruction

stsx (Store String Indexed) Instruction

Understanding Fixed Point Address Computation Instructions

There are three fixed point instructions for address computation.

cal (Compute Address Lower) Instruction

cau (Compute Address Upper) Instruction

cax (Compute Address) Instruction

Understanding Fixed Point Arithmetic Instructions

The Fixed Point Arithmetic Instructions treat the contents of registers as 32-bit signed integers.

a (Add) Instruction

abs (Absolute) Instruction

ae (Add Extended) Instruction

ai (Add Immediate) Instruction

ai. (Add Immediate and Record) Instruction

ame (Add To Minus One Extended) Instruction

aze (Add To Zero Extended) Instruction

div (Divide) Instruction

divs (Divide Short) Instruction

doz (Difference or Zero) Instruction

dozi (Difference or Zero Immediate) Instruction

mul (Multiply) Instruction

muli (Multiply Immediate) Instruction

muls (Multiply Short) Instruction

nabs (Negative Absolute) Instruction

neg (Negate) Instruction

sf (Subtract From) Instruction

sfe (Subtract From Extended) Instruction

sfi (Subtract From Immediate) Instruction

sfme (Subtract From Minus One Extended) Instruction

sfze (Subtract From Zero Extended) Instruction

si (Subtract Immediate) Instruction

si. (Subtract Immediate and Record) Instruction

Understanding Fixed Point Logical Instructions

The Logical Instructions perform the indicated operations in a bit-wise fashion.

and (AND) Instruction

andc (AND With Complement) Instruction

andil. (AND Immediate Lower) Instruction

andiu. (AND Immediate Upper) Instruction

cntlz (Count Leading Zeros) Instruction

eqv (Equivalent) Instruction

exts (Extend Sign) Instruction

nand (NAND) Instruction

nor (NOR) Instruction

or (OR) Instruction

orc (OR With Complement) Instruction

oril (OR Immediate Lower) Instruction

oriu (OR Immediate Upper) Instruction

xor (XOR) Instruction

xoril (XOR Immediate Lower) Instruction

xoriu (XOR Immediate Upper) Instruction

Understanding Fixed Point Rotate Instructions

The Fixed Point Processor performs rotate operations on data from a General Purpose Register. The result of the rotate with mask instructions is either inserted into the register under control of the provided mask or ANDed with the mask before the result is placed in the

register. The rotate operations move a specified number of bits to the left. The bits that exist from bits position 0 enter at bit position 31.

When the rotate with insert is used, the result of the rotate operation is placed into the target General Purpose Register under control of the provided mask. If a mask bit is 1, then the associated bit of the rotated data (0 or 1) is placed into the target General Purpose Register. If the mask bit is 0, the associated data bit (0 or 1) from the register remains unchanged.

The rotate left instructions allow rotate right instructions to be performed (in concept) by a rotate left of 32–N bits, where N is the number of positions to rotate right.

Fixed Point Bit Mask Instructions

maskg (Mask Generate) Instruction

maskir (Mask Insert From Register) Instruction

rrib (Rotate Right And Insert Bit) Instruction

Fixed Point Rotate With Mask Instructions

rlimi (Rotate Left Immediate Then Mask Insert) Instruction

rlinm (Rotate Left Immediate Then AND With Mask) Instruction

rlmi (Rotate Left Then Mask Insert) Instruction

rlnm (Rotate Left Then AND With Mask) Instruction

Understanding Fixed Point Shift Instructions

The Fixed Point Shift Instructions logically perform left and right shifts. The result of a shift instruction is placed in a General Purpose Register under control of a generated mask.

When the result of a shift instruction is placed into register *RA*, the target register, under the control of a generated mask, one of the following occurs:

- If the mask bit is a 1, the respective bit from either the rotated word or a word of zeroes is placed into the target General Purpose Register.
- If the mask bit is a 0, the corresponding bit from either the Multiply Quotient Register or a word of 32 sign bits from the source General Purpose Register is placed into the target General Purpose Register.

Setting the instruction's Record bit to 1 sets bits in the Condition Register according to the value of the contents of the target General Purpose Register at the completion of the instruction. The Condition Register is a set as if a compare between the contents of the target General Purpose Register and the value zero had been performed.

sl (Shift Left) Instruction

sle (Shift Left Extended) Instruction

sleq (Shift Left Extended With MQ) Instruction

sliq (Shift Left Immediate With MQ) Instruction

slliq (Shift Left Long Immediate with MQ) Instruction

sllq (Shift Left Long With MQ) Instruction

slq (Shift Left With MQ) Instruction

sr (Shift Right) Instruction

sra (Shift Right Algebraic) Instruction

srai (Shift Right Algebraic Immediate) Instruction

sraiq (Shift Right Algebraic Immediate With MQ) Instruction

sraq (Shift Right Algebraic With MQ) Instruction

sre (Shift Right Extended) Instruction

srea (Shift Right Extended Algebraic) Instruction

sreq (Shift Right Extended With MQ) Instruction

sriq (Shift Right Immediate With MQ) Instruction

srliq (Shift Right Long Immediate With MQ) Instruction

srlq (Shift Right Long With MQ) Instruction

srq (Shift Right With MQ) Instruction

Understanding Fixed Point Move To/From System Registers Instructions

Several instructions move the contents of one system register into another system register or into a General Purpose Register.

mcrxr (Move To Condition Register From XER) Instruction

mfcr (Move From Condition Register) Instruction

mfmsr (Move From Machine State Register) Instruction

mfspr (Move From Special Purpose Register) Instruction

mtcrf (Move To Condition Register Fields) Instruction

mtspr (Move To Special Purpose Register) Instruction

Related Information

See the following articles in *POWERstation and POWERserver Hardware Technical Reference — General Information*: Condition Register, Machine State Register , General Purpose Registers, Fixed Point Exception Register, Multiply Quotient Register.

Branch Processor Overview on page 1–3.

Floating Point Processor Overview

The Floating Point Processor instructions are provided to perform arithmetic operations in floating point registers and move floating point data between storage and these registers.

Understanding Floating Point Numbers

A floating point number consists of a signed exponent and a signed significand, and expresses a quantity that is the product of the significand and the number 2**exponent. Encodings are provided in the data format to represent:

- Finite numeric values
- +– Infinity
- Values which are "Not a Number" (NaN)

Operations involving infinities produce results obeying traditional mathematical conventions. NaNs have no mathematical interpretation. Their encoding permits a variable diagnostic information field. They may be used to indicate uninitialized variables and can be produced by certain invalid operations.

Understanding Floating Point Processor Registers

Floating Point Registers

There are thiry–two 64-bit Floating Point Registers, numbered from Floating Point Register 0–31. All Floating Point instructions provide a 5-bit field that is used to specify which Floating Point Registers are to be used in the execution of the instruction. Every instruction that interprets the contents of a Floating Point Register as a floating point value uses the double precision floating point format for this interpretation.

All floating point instructions other than loads and stores are performed on operands located in Floating Point Registers and place the results in a Floating Point Register. The Floating Point Status and Control Register and the Condition Register maintain status information about the outcome of some floating point operations.

Double and Single Format Conversions on Load and Store

Load and store double instructions transfer 64 bits of data without conversion between storage and a Floating Point Register in the Floating Point Processor. Load single instructions convert a stored single floating format value to the same value in double floating format and transfer that value into a Floating Point Register. Store single instruction, do the opposite, converting a double floating format value in a Floating Point Register into a single floating format value, prior to storage.

Understanding The Floating Point Status and Control Register

The Floating Point Status and Control Register is a 32-bit register that contains control flags which govern the handling of Floating Point Exceptions (bits 20–31) and record information about the results of floating point operations (bits 0–19).

Understanding Floating Point Load Instructions

There are load instructions for single precision and double precision. Double precision data is loaded directly into a Floating Point Register. Since Floating Point Registers only support floating point double precision operands, the processor converts single precision data to double precision prior to loading.

lfd (Load Floating Point Double) Instruction

lfdu (Load Floating Point Double with Update) Instruction

lfdux (Load Floating Point Double with Update Indexed) Instruction

lfdx (Load Floating Point Double Indexed) Instruction

lfs (Load Floating Point Single) Instruction

lfsu (Load Floating Point Single with Update) Instruction

lfsux (Load Floating Point Single with Update Indexed) Instruction

lfsx (Load Floating Point Single Indexed) Instruction

Understanding Floating Point Store Instructions

There are store instructions for single precision and double precision. Single precision stores convert Floating Point Register contents to single precision proir to storage.

stfd (Store Floating Point Double) Instruction

stfdu (Store Floating Point Double With Update) Instruction

stfdux (Store Floating Point Double With Update Indexed) Instruction

stfdx (Store Floating Point Double Indexed) Instruction

stfs (Store Floating Point Single) Instruction

stfsu (Store Floating Point Single With Update) Instruction

stfsux (Store Floating Point Single With Update Indexed) Instruction

stfsx (Store Floating Point Single Indexed) Instruction

Understanding Floating Point Move Instructions

The Floating Point Move Instructions copy data from one Floating Point Register to another. Data may be modified depending upon the instruction.

fabs (Floating Absolute Value) Instruction

fmr (Floating Move Register) Instruction

fnabs (Floating Negative Absolute Value) Instruction

fneg (Floating Negate) Instruction

Understanding Floating Point Arithmetic Instructions

The Floating Point Arithmetic Instructions perform arithmetic functions using floating point data.

fa (Floating Add) Instruction

fd (Floating Divide) Instruction

fm (Floating Multiply) Instruction

frsp (Floating Round to Single Precision) Instruction

fs (Floating Subtract) Instruction

Understanding Floating Point Accumulate Instructions

The Floating Point Accumulate Instructions combine a multiply and an add operation without an intermediate rounding operation.

fma (Floating Multiply Add) Instruction

fms (Floating Multiply Subtract) Instruction

fnma (Floating Negative Multiply Subtract) Instruction

fnms (Floating Negative Multiply Subtract) Instruction

Understanding Floating Point Compare Instructions

There are two instructions for performing ordered and unordered compares of the contents of two Floating Point Registers. You specify which field in the Condition Register receives the result of the compare.

fcmpo (Floating Compare Ordered) Instruction

fcmpu (Floating Compare Unordered) Instruction

These instructions set one bit in the field to one, and the others to zero. The compare result bits have the following interpretations:

Bit 0	(*FRA*) < (*FRB*)	Less Than
Bit 1	(*FRA*) > (*FRB*)	Greater Than
Bit 2	(*FRA*) = (*FRB*)	Equal
Bit 3	(*FRA*) ? (*FRB*)	Unordered

Understanding Floating Point Status and Control Register Instructions

These instructions manipulate data in the Floating Point Status and Control Register.

mcrfs (Move to Condition Register from FPSCR) Instruction

mffs (Move from FPSCR) Instruction

mtfsf (Move to FPSCR Fields) Instruction

mtfsfi (Move to FPSCR Fields Immediate) Instruction

mtfsb1 (Move to FPSCR Bit 1) Instruction

mtfsb0 (Move to FPSCR Bit 0) Instruction

Related Information

The **frsp** (Floating Round to Single Precision) instruction, **mtcrf** (Move to Condition Register Fields) instruction, **mtfsf** (Move to FPSCR Fields) instruction, **mtfsfi** (Move to FPSCR Fields Immediate) instruction, **mtfsb1** (Move to FPSCR Bit 1) instruction, **mtfsb0** (Move to FPSCR Bit 0) instruction.

See the following articles in *POWERstation and POWERserver Hardware Technical Reference — General Information*: Floating Point Data Representation, Floating Point Resource Management, Floating Point Exceptions, Machine State Register.

Chaper 2. Syntax and Semantics

Syntax and Semantics Overview

This overview explains the syntax and semantics of assembler language, including The Character Set, Reserved Words, Line Format, Statements, Symbols, Constants, Operators, and Expressions.

Understanding The Character Set

All letters and numbers are allowed. The assembler discriminates between uppercase and lowercase letters. To the assembler, the symbols *Name* and *name* identify distinct symbols.

Some blank spaces are required, while others are optional. The assembler allows you to substitute tabs for spaces.

The following characters have special meaning in RISC System/6000 assembler language.

, (comma)

Operand separator. Commas are allowed in statements only between operands.

Example:

```
a 3,4,5
```

(pound sign)

Comments. Anything after the # to the end of the line is ignored by the assembler. A # can be the first character in a line, or it can be preceded by any number of characters, blank spaces, or both.

Example:

```
a 3,4,5 # Puts the sum of GPR4 and GPR5 into GPR3.
```

: (colon)

Defines a label. The : always appears immediately after the last character of the label name and defines a label equal to the value contained in the location counter at the time the assembler encounters the label.

Example:

```
add: a 3,4,5    # Puts add equal to the address where the a
                # instruction is found.
```

; (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:

```
a 3,4,5         # These two lines have
a 4,3,5         # the same effect as...
a 3,4,5; a 4,3,5        # ...this line.
```

$ (dollar sign)

Refers to the current value in the assembler's current location counter.

Example:

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

Understanding Reserved Words

There are no reserved words in the RISC System/6000 Microprocessor assembler language. The mnemonics for instructions and pseudo-ops are not reserved and can 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.

Understanding Line Format

The RISC System/6000 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 a code line is longer than one line on a terminal, line wrapping will depend on the editor used. However, the listing wiil only display 100 ASCII characters per line.

Blank lines are allowed; the assembler ignores them.

Understanding Statements

The RISC System/6000 Microprocessor assembler language has three kinds of statements: instruction statements, pseudo-operation statements, and null statements. The Assembler also uses Separator Characters, Labels, Mnemonics, Operands, and Comments.

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 appears:

- An ASCII new–line character
- A comment character (#)
- A semicolon(;).

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 state is on a different line. The assembler gives the label the value contained in the current location counter. For example,

```
here:
        a 3,4,5
```

is synonymous with

```
here:   a 3,4,5
```

Note: Certain pseudo-ops may prevent a null statement's label from receiving the value of the address of the next statement.

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. 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.

Example:

```
subtr:    sf 3,4,5

# The label subtr: receives the value
# of the address of the sf instruction.
# You can now use subtr 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:

```
# Assume that the location counter now
# contains the value of 98.

place:    .long expr

# When the assembler processes 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 **ai** (Add Immediate) or **sf** (Subtract From). This mnemonic describes an operation where the RISC System/6000 Microprocessor processes a single machine instruction that is associated with a numerical operation code (opcode). All instructions are 4 bytes long. When the assembler encounters an instruction, 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 opcode, 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.

Understanding 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, uppercase or lowercase letters, or any combination of these. The symbol cannot contain any blanks or special characters, and cannot begin with a digit. Uppercase and lowercase letters 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. Also note that other routines linked to the assembler language files may have their own constraints on symbol length.

With the exception of .csect or TOC entry names, symbols may be used to represent storage locations or arbitrary data. The value of a symbol is always a 32-bit quantity.

The following are valid symbol names:

```
READER

XC2345

result.a

resultA

balance_old

_label9

.myspot
```

The following are not possible symbol names:

`7_sum` (begins with a digit)

`#ofcredits` (the # makes this a comment)

`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**, **.lcomm**, **.dsect**, or **.csect** pseudo-op.

Defining a symbol with a Label

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

Example:

```
                .using          dataval[RW],5
loop:
                bgt             cont
  .
  .
                bdz             loop
cont:           l               3,dataval
                a               4,3,4
  .
  .
.csect dataval[RW]
dataval:        .short    10
```

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

At runtime, an address is calculated from *dataval*, the offset, and GPR 5, which needs to contain the address of **csect dataval[RW]**. In the example above, the **l** 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 of the location represented by *dataval*.

Defining a Symbol with a Pseudo-op

Use a symbol as the name operand of a **.set** pseudo-op to define the symbol. This pseudo-op has the format:

```
.set  name,exp
```

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.

For example:

```
        .set            number,10
  .
  .
        ai      4,4,number
```

In the example above, the object code for the **ai** 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. Also, because the symbol occupies no storage, you cannot change the value of the symbol at runtime; reassembling the file will give the symbol a new value.

A symbol can also be defined by using it as the *name* operand of a **.comm**, **.lcomm**, **.csect** or, **.dsect** pseudo-op. Except for **.dsect**, the value assigned to the symbol does describe storage space.

CSECT and TOC Entry Names

A symbol can also be defined when used as the *qualname* operand of the **.csect** or **.tc** pseudo-op. When used in this context, the symbol is defined as the name of a csect or a TOC entry with the specified storage class. Therefore, the storage class qualifier is *required* when naming csects or TOC entries. Once defined, the symbol takes on a storage class that corresponds to the name qualifier.

For csects, different csects can have the same name but different storage classes; therefore, the storage class identifier must be used when referring to a csect name as an operand of other pseudo-ops. However, this is not the case with the binder. If csect names are externalized, establish unique names for the externalized csects. A csect operand name takes the form of :

```
symbol[XX]
```

or

```
symbol{XX}
```

where the required square brackets([]]) or curly ({ }) brackets both produce the same results and surround a two–character storage class identifier which can be one of the following:

PR	RO
DB	GL
XO	SV
RW	UA
DS	TI
TB	LC
TC	TC0

in uppercase or lowercase. The following example illustrates the definition and a possible use of a csect:

```
     .csect prog1data[rw]
# Defines a csect called "prog1data"
# of storage class 'RW'.
  .
  .
       .long prog1data[RW]
```

For TOC entries , different TOC entries can have the same name but will be considered the same TOC entry. The storage class identifier must be used when using the TOC entry name as an operand. A TOC entry operand name takes the form of:

```
          symbol[TC]
```

where *TC* stands for TOC entry. The following example illustrates the definition and a possible use of a TOC entry name:

```
     .tc p1data[TC],p1data[RW]
   .
   .
      .long p1data[TC]
```

The Special Symbol TOC

Provisions have been made for the special symbol TOC. In XCOFF format modules, this symbol is reserved for the TOC anchor, or the first entry in the TOC. The symbol TOC has been predefined in the assembler so that the symbol TOC can be referred to if its use is required. The **.toc** pseudo–op creates the TOC anchor entry. For example, the following data declaration declares a word that contains the address of the beginning of the TOC.

```
         .long TOC[TC0]
```

This symbol is undefined unless a **.toc** pseudo–op is contained within the assembler file.

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.

```
# Assume that GPR 6 contains the address of .csect data[RW].
      l 5,ten(6)
        .
        .
  .csect data[RW]
  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 undefined symbols, it gives an error message. External symbols may be declared in an **.extern** statement.

Declaring an External Symbol

If a local symbol is used that is defined in another module, the **.extern** pseudo-op is used to declare that symbol in the local file as an external symbol. Any undefined symbols that do not appear in an **.extern** or **.globl** statement will be flagged with an error.

Understanding Constants

The RISC System/6000 Microprocessor assembler language provides four kinds of constants:

- Arithmetic constants
- Character constants
- Symbolic constants
- String 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. When the assembler encounters a symbol being used as a constant, the value of the symbol is assembled into the instruction.

Arithmetic Constants

There are four kinds of arithmetic constants:

- Decimal
- Octal
- Hexadecimal
- Floating Point

The largest signed positive integer number that can be represented is the decimal value 2**31–1. The largest negative value is –231. Regardless of the base (e.g., decimal, hexadecimal or octal), the as assembler regards integers as 32-bit constants.

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 5,4,10
# Add the decimal value 10 to the contents
# of GPR 4 and put the result in GPR 5.
```

Do not prefix decimal numbers with a zero. A leading zero indicates that the number is octal.

Octal Constants

To specify that a number is octal, prefix the number with the *numeral* 0.

```
ai 5,4,0377
# Add the octal value 0377 to the contents
# of GPR 4 and put the result in GPR 5.
```

Hexadecimal Constants

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

```
ai 5,4,0xF
# Add the hexadecimal value 0xF to the
# contents of GPR 4 and put the result
# in GPR 5.
```

Floating Point Constants

A floating point constant has the following components in order.

Integer Part	Must be one or more digits.
Decimal Part	Optional
Fraction Part	Must be one or more digits.
Exponent Part	Optional. Consists of an **e** or **E**, possibly followed by a + or –, followed by one or more digits.

For assembler input, you may omit the fraction part. For example, the following are valid floating point constants.

```
0.45
1e+5
4E–11
0.99E6
357.22e12
```

Floating point constants are only allowed wherever **fcon** expressions are found.

There is no bounds checking for the operand.

Note: The **atof** subroutine is called to get the floating point number from input. Check current documentation for restrictions and return values.

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.

```
cal 3,'X(0)
# Loads GPR 3 with the ASCII code for
# the character X (that is, 0x58).

# After the cal instruction executes, GPR 3 will
# contain binary
# 0x0000 0000 0000 0000 0000 0000 0101 1000.
```

Symbolic Constants

A symbol can be used as a constant. A symbol can be given a value so that the value can be referred to by name, instead of using the value itself.

Using a symbol as a constant is convenient if a value occurs frequently in a program. Define the symbolic constant once by giving the value a name. To change its value, simply change the definition, not every reference to it in the program. The changed file must be reassembled before the symbol constant is valid.

A symbolic constant can be defined by using it as a label or by using it in a **.set** statement.

Strings

String constants are different than other types of constants in that they can only be used as operands to certain pseudo–ops, such as **.rename**, **.byte**, or **.string** pseudo–ops.

The syntax of string constants are any number of characters enclosed in double quotes (").

```
"any number of characters"
```

The double quote character is obtained with two double quotes.

```
"a double quote character is specified like this "" "
```

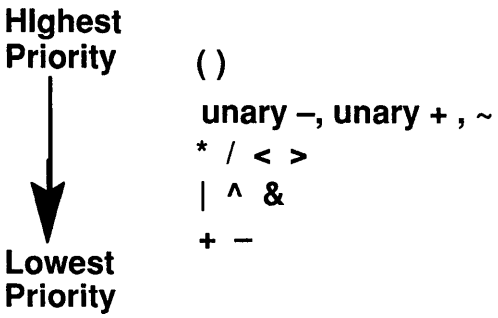
Understanding Operators

The RISC System/6000 Microprocessor assembler language provides the following operators.

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

Operator Precedence

Operator precedence for 32-bit expressions is shown in the following figure.



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	Remainder
8/3	2	2
8/–3	–2	2
(–8)/3	–2	–2
(–8)/(–3)	2	–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 the result to newdata.
```

Understanding 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, and then uses that value as an operand. Expressions have a type as well as a value.

There are five types of expressions.

- Absolute Expressions
- Relocatable Expressions
- External Expressions
- Restricted External Expressions
- TOC Relative Expressions

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.

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.

An absolute expression is one of the following:

- A 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 "relocatable's" refer to or are contained within the same assembler csect.

The definitions of "absolute" and "relocatable" given above are recursive. For example,

```
absolute<operator>absolute<operator>relocatable, – relocatable
```

is a valid expression.

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

Relocatable Expressions

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

Since the csects can be relocated independently, the type of a relocatable expression includes the csect that contains it.

A relocatable expression is one of the following:

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

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

External Expressions

External expressions refer to external symbols (symbols not defined, but declared 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**.

An external expression is one of the following:

- A symbol declared with **.comm**
- A symbol declared with **.extern**
- An undefined symbol declared with **.globl**
- external + absolute
- external – absolute
- absolute + external
- absolute – external

The definitions of "absolute" and "external" 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 expression are:

```
external+relocatable
external+external
```

Restricted External Expressions

A restricted form of external expression allows external expressions to be combined in a non–recursive fashion. The following expressions are allowed and produce restricted external (*rext*) expression:

- external–external
- external–relocatable
- relocatable–external

Restricted expressions can be formed recursively when used with absolute expressions as follows:

- rext–abs
- rext+abs

Invalid restricted expressions are:

- rext–rext
- rext–external
- relocatable–rext
- trel – trel

TOC Relative Expressions

One other class of expressions has been provided that allows "toc" relative expressions. Expressions of this type are always relative to the beginning of the table of contents or "toc" of a file. Valid TOC relative (**trel**) expressions are:

- Labels on **.tc** entries
- trel + abs
- trel – abs

Related Information

The **atof** subroutine in *General Programming Concepts.*

The **.comm** pseudo–op, **.csect** pseudo–op, **.double** pseudo–op, **.dsect** pseudo–op, **.float** pseudo–op, **.lcomm** pseudo–op, **.tc** pseudo–op, **.toc** pseudo–op, **.tocof** pseudo–op.

Chapter 3. Addressing

Addressing Overview

Discusses Addressing modes including absolute addressing, absolute immediate addressing, relative immediate addressing, explicit based addressing, and implicit Based addressing.

Understanding Absolute Addressing

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 current instruction address.

Both the branch conditional register (**bcr**) instruction and the branch conditional to count register (**bcc**) instruction use an absolute addressing mode. However, the register is not an operand but a specific register, namely the link register (for the **bcr** instruction) or the count register (for the **bcc** instruction). These registers must be loaded prior to instruction execution.

Understanding Absolute Immediate Addressing

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 current instruction address.

Both the branch absolute (**ba**) instruction and the branch conditional absolute (**bca**) instruction use an absolute immediate addressing mode. These instructions assemble a 26–bit immediate operand which is divided by four to become the branch target address. The immediate operand can be an absolute, a relocatable, or an external expression.

Understanding Relative Immediate Addressing

Relative immediate addresses are specified as immediate data within the object code and are calculated relative to the current instruction location. All the instructions that use relative immediate addressing are branch instructions. These instructions have immediate data that is the displacement in fullwords from the current instruction location. At execution, the immediate data is sign extended, logically shifted to the left two bits, and added to the address of the branch instruction to calculate the branch target address.

Understanding Explicit Based Addressing

Programmers can write an explicit based address by specifying a base register number, *RA*, and a displacement, *D*. 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. If an instruction does not have an operand form of *D*(*RA*), then the instruction cannot have an explicit based address.

Although programmers must use an absolute expression to specify the base register itself, the contents of the base register can be specified by an absolute, a relocatable, or an 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.

When using explicit based addressing, remember that:

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 *D* 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.

Understanding Implicit Based Addressing

To specify an implicit based address as an operand for an instruction, omit the *RA* 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 the base were explicitly specified in the instruction.

Implicit based addresses can be relocatable or absolute, depending on the type of expression used to specify the contents of *RA* at runtime. Usually, programmers specify the contents of *RA* 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 *D*(*RA*) still points to the correct address after relocation.

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

When using implicit based addressing:

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** pseudo–op, the assembler will use this base register value to process all instructions that require a based address.
3. Load each base register with the previously specified value.

When the (*RA*) operand is omitted, the *D* operand remains. *D* is a label, an absolute expression, or an expression containing a label.

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

```
.toc
T.data: .tc data[tc],data[rw]
.csect data[rw]
        foo: .long 2,3,4,5,6
        bar: .long 777

        .csect text[pr]
        .align 2
        l 10,T.data(2)  # Loads the address of
                                # csect data[rw] into GPR 10.
        .using data[rw], 10     # Specify displacement.
        l 3,foo         # The assembler generates l 3,0(10)
        l 4,foo+4               # The assembler generates l 4,4(10)
        l 5,bar         # The assembler generates l 5,20(10)
```

Understanding the Location Counter

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

Related Information

The **bcc** (Branch Conditional to Count Register) instruction, **bcr** (Branch Conditional Register) instruction, **ba** (Branch Absolute) instruction, **bca** (Branch Conditional Absolute) instruction.

The **.using** pseudo–op, **.drop** pseudo–op.

Understanding Branch Instructions on page 1–3.

Branch Processor Overview on page 1–3.

Chapter 4. Assembling, Linking, and Running

Assembling, Linking, and Running A Program Overview

Understanding Assembler 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 performs the following tasks:

- Allocates space for instructions and storage areas you request
- Fills in the values of constants, where possible
- 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 line of the source file at a time. If this source statement has a valid symbol in the label field, the assembler checks to make sure 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 reassigns 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, XO format). The assembler then 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 (preceded by #) 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 has been associated with a location counter value in the symbol table. When there are no more source statements to read, the second pass starts at the beginning of the program again.

The Second Pass

On the second pass the assembler:

- Examines the operands for symbolic references to storage locations, and resolves these symbolic references using information in the symbol table.
- Translates source statements into machine code and constants, thus filling the allocated space with object code.
- Produces a file containing error messages, should any have occurred.

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 the symbol was never the subject of a **.comm**, **.csect, .lcomm**, **.sect**, or **.set** pseudo–op.

This could be either a deliberate external reference or an accidental programmer error, such as misspelling a symbol name. The assembler indicates an error. All external references must appear in a **.extern** or **.globl** statement.

The assembler logs errors such as incorrect data alignment. However, many possible alignment problems are indicated with warning statements that will not halt assembly. The **–w** flag must be used to display these warning messages.

After the programmer corrects assembly errors, the program is ready to be linked.

Assembling and Linking with the cc Command

A simpler way to assemble and link a program is to use the **cc** command to link the files as follows:

```
cc pgm.o subs1.o subs2.o
```

Since the **cc** command automatically uses the link options and necessary support libraries, you do not need to specify them on the command line (the configuration file **cc.cfg** provides this information). For this reason, use the **cc** command to link files when producing programs that run under the AIX operating system. If already assembled, the object file must have a **.o** extension as indicated in the previous example. The **cc** command will invoke the assembler on files that have a **.s** extension. Flags used with the **as** command can also be directed to the assembler through the **cc** command. To produce a listing and object file:

```
cc –c –Wa,–l   files.s
```

Warning: The **cc** command invokes the assembler and then continues processing normally. Therefore,

```
cc –Wa,–l,–oXfile.o file.s
```

will produce an object file *Xfile*.**o** and a listing *file*.**lst** from the assembler, but then will continue processing by calling the **ld** command with *file*.**o**. This will fail because the assembler produced *Xfile*.**o**.

Interpreting an Assembler Listing

The **–l** flag on the **as** command produces a listing of an assembler language file.

Assume that a programmer wants to display the words "hello, world." The C program would appear like the following example:

```
main ( )
{
   printf ("hello, world\n");
}
```

Assembling **hello.s** with the following command produces the assembler program listing.

```
as –l hello.s
```

This produces an output file named **hello.lst**. The complete listing for **hello.lst** is given below.

```
 hello.s                                  03/28/90

File# Line#  Name    Loc Ctr   Object Code       Source

0      1 |                                      ##############################
0      2 |                                      # C source code
0      3 |                                      ##############################
0      4 |                                      #   hello()
0      5 |                                      #   {
0      6 |                                      #    printf("hello,world\n");
0      7 |                                      #   }
0      8 |                                      ##############################
0      9 |                                      #   Compile as follows:
0     10 |                                      #   cc -o helloworld hello.s
0     11 |                                      #
0     12 |                                      ##############################
0     13 |                                       .file  "hello.s"
0     14 |                                      #Static data entry in
0     15 |                                      #T(able)O(f)C(ontents)
0     16 |                                       .toc
0     17 | data   00000000 00000040          T.data:   .tc data[tc],data[rw]
0     18 |                                       .globl main[ds]
0     19 |                                      #main[ds] contains definitions for
0     20 |                                      #runtime linkage of function main
0     21 |                                       .csect   main[ds]
0     22 | main   00000000 00000000            .long    .main[PR]
0     23 | main   00000004 00000050            .long    TOC[tc0]
0     24 | main   00000008 00000000            .long    0
0     25 |                                      #Function entry in
0     26 |                                      #T(able)O(f)C(ontents)
0     27 |                                       .toc
0     28 | .main  00000000 00000034          T.hello: .tc .main[tc],main[ds]
0     29 |                                       .globl   .main[PR]
0     30 |
0     31 |                                      #Set routine stack variables
0     32 |                                      #Values are specific to
0     33 |                                      #the current routine and can
0     34 |                                      #vary from routine to routine
0     35 |                         00000020         .set  argarea,    32
0     36 |                         00000018         .set  linkarea,   24
0     37 |                         00000000         .set  locstckarea, 0
0     38 |                         00000001         .set  ngprs,      1
0     39 |                         00000000         .set  nfprs,      0
0     40 |                         0000003c         .set  szdsa,
8*nfprs+4*ngprs+linkarea+argarea+locstckarea
0     41 |
0     42 |                                      #Main routine
0     43 |                                       .csect   .main[PR]
0     44 |
0     45 |
0     46 |                                      #PROLOG: Called Routines
```

```
0    47 |                                  #       Responsibilities
0    48 |                                  #Get link reg.
0    49 | .main  00000000  7c0802a6          mflr  0
0    50 |                                  #Not required to Get/Save CR
0    51 |                                  #because current routine does
0    52 |                                  #not alter it.
0    53 |
0    54 |                                  #Not required to Save FPR's
0    55 |                                  #14–31 because current routine
0    56 |                                  #does not alter them.
0    57 |
0    58 |                                  #Save GPR 31.
0    59 | .main  00000004  bfe1fffc          stm   31, –8*nfprs–4*ngprs(1)
0    60 |                                  #Save LR if non–leaf routine.
0    61 | .main  00000008  90010008          st    0, 8(1)
0    62 |                                  #Decrement stack ptr and save
0    63 |                                  #back chain.
0    64 | .main  0000000c  9421ffc4          stu   1, –szdsa(1)
0    65 |
0    66 |
0    67 |                                  #Program body
0    68 |                                  #Load static data address
0    69 | .main  00000010  81c20000          l     14,T.data(2)
0    70 |                                  #Line 3, file hello.c
0    71 |                                  #Load address of data string
0    72 |                                  #from data addr.
0    73 |                                  #This is a parameter to printf()
0    74 | .main  00000014  386e0000          cal   3,_helloworld(14)
0    75 |                                  #Call printf function
0    76 | .main  00000018  4bfffe9           bl    .printf[PR]
0    77 | .main  0000001c  4def7b82          cror  15, 15, 15
0    78 |
0    79 |
0    80 |                                  #EPILOG: Return Sequence
0    81 |                                  #Get saved LR.
0    82 | .main  00000020  80010044          l     0, szdsa+8(1)
0    83 |
0    84 |                                  #Routine did not save CR.
0    85 |                                  #Restore of CR not necessary.
0    86 |
0    87 |                                  #Restore stack ptr
0    88 | .main  00000024  3021003c          ai    1, 1, szdsa
0    89 |                                  #Restore GPR 31.
0    90 | .main  00000028  bbe1fffc          lm    31, –8*nfprs–4*ngprs(1)
0    91 |
0    92 |                                  #Routine did not save FPR's.
0    93 |                                  #Restore of FPR's not necessary.
0    94 |
0    95 |                                  #Move return address
0    96 |                                  #to Link Register.
0    97 | .main  0000002c  7c0803a6          mtlr  0
0    98 |                                  #Return to address
0    99 |                                  #held in Link Register.
0   100 | .main  00000030  4e800021          brl
```

```
0   101 |
0   102 |
0   103 |                                #External variables
0   104 |                                .extern  .printf[PR]
0   105 |
0   106 |                                ##############################
0   107 |                                # Data
0   108 |                                ##############################
0   109 |                                #String data placed in
0   110 |                                #static csect data[rw]
0   111 |                                 .csect  data[rw]
0   112 |                                 .align  2
0   113 |                                _helloworld:
0   114 | data  00000000  68656c6c         .byte  0x68,0x65,0x6c,0x6c
0   115 | data  00000004  6f2c776f        .byte  0x6f,0x2c,0x77,0x6f
0   116 | data  00000008  726c640a         .byte  0x72,0x6c,0x64,0xa,0x0
        | data  0000000c  00
```

Assembler Listing Headings

The first line of the assembler listing gives two peices of information.

1. The name of the source file – in this case, **hello.s**.
2. The date the listing file was created – in this case, 03/28/90.

The assembler listing is given in six columns. The column headings are as follows.

1. File# – Source file number. Files included with the M4 macro processor (–I option) will display by number the file in which the statement was found.
2. Line# – Refers to the line number of the assembler source code.
3. Name – The name of the csect where this line of source code originates
4. Loc Ctr – The value contained in the assembler's location counter. The listing only shows a location counter value for those assembler language instructions that generate object code.
5. Object Code – Shows the hexadecimal representation of the object code generated by each line of the assembler program. Since each instruction is 32 bits, each line in the assembler listing shows a maximum of 4 bytes. Any remaining bytes in a line of assembler source code are shown on the following line(s).
6. Source – The assembler source code for the program. A limit of 100 ASCII characters will be displayed per line.

Subroutine Linkage Convention

The subroutine linkage convention describes the machine state at subroutine entry and exit. When followed, this scheme allows routines that are compiled separately in the same or different languages to be linked and executed when called.

The linkage convention allows for parameter passing and return values to be in FPRs, GPRs, or both. (GPRs are also referred to as registers.)

Register Usage

To be compatible with other programming languages, called and calling routines must observe certain conventions on registers usage. There are two types of registers of importance in this regard:

- A *volatile* register — holds a value on entry that need not be preserved when the called routine returns.
- A *non-volatile* register — holds a value on entry that must be preserved on exit from the calling routine.

If the value of a non-volatile register changes across the call, then the called routine must:

- Save the value of the register before the register is change
- Restore the original value of the register before returning to the calling routine.

If a register is not designated as saved during the call, its contents may be changed during the call. Conversely, if a register is saved, its contents must be preserved across the call.

Note: The assembler generates a program adhering to the XCOFF format. Using certain general purpose registers indiscriminately can lead to unpredictable results. For reference, note the General Purpose Register usage conventions specified in the following table.

General Purpose Register	Preserved Across Calls	Status	Convention
0	no	Scratch	Used in prologs
1	yes	Saved	Stack Pointer
2	yes	Saved	TOC
3	no	Scratch	1st word of argument list; return value
4	no	Scratch	2nd word of argument list; return value
5	no	Scratch	3rd word of argument list; return value
6	no	Scratch	4th word of argument list; return value
7	no	Scratch	5th word of argument list; return value
8	no	Scratch	6th word of argument list; return value
9	no	Scratch	7th word of argument list; return value
10	no	Scratch	8th word of argument list; return value
11	no	Scratch	Scratch; Pointer to FCN; DSA pointer to int proc (Env)
12	no	Scratch	PL8 exception return
13–31	yes	Saved	Non-volatile

The following table lists floating–point registers and their functions. The floating–point registers are double precision (64 bits).

Floating Point Register	Preserved Across Calls	Use
0	no	
1	no	FP parameter 1, function return 1.
2	no	FP parameter 2, function return 2.
.	.	.
.	.	.
.	.	.
13	no	FP parameter 13, function return 13.
14–31	yes	

The following table lists special purpose register conventions.

Special Purpose Register	Preserved Across Calls
Condition Register	
Bits 0–7 (CR0,CR1)	no
Bits 8–19 (CR0,CR1)	yes
Bits 20–23 (CR0,CR1) Reserved for system use. Never set or changed.	yes
Bits 24–31 (CR0,CR1)	no
Link Register	no
Count Register	no
MQ Register	no
XER Register	no
FPSCR Register	no

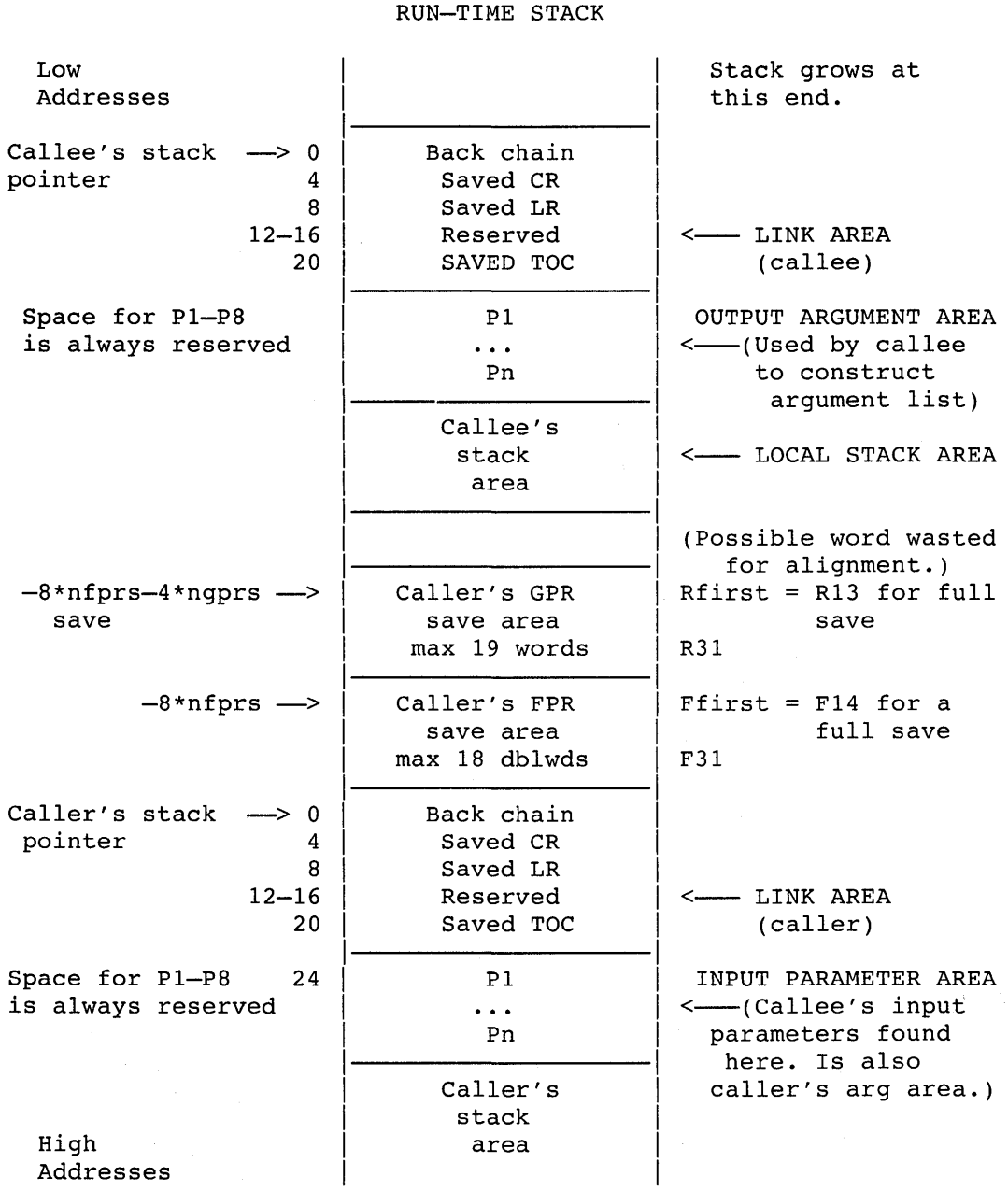
Stack

The stack is a portion of storage that is used to hold local storage, register save areas, parameter lists, and call chain data. The stack grows from higher addresses to lower addresses. A stack pointer register (register 1) is used to mark the current "top" of the stack.

A stack frame is the portion of the stack used by a single procedure. You can consider the input parameters as being part of the current stack frame. In a sense, each output argument belongs to both the caller's and the callee's stack frames. In either case, the stack frame size is best defined as the difference between the caller's stack pointer and the callee's.

The storage map of a typical stack frame is shown below. In the diagram, the current routine has acquired a stack frame which allows it to call other functions. If no calls are made, and there are no local variables or temps, then the function need not allocate a stack frame. It can still use the register save area at the top of the caller's stack frame, if needed.

The stack frame is double word aligned. The FPR save area and the parameter area (P1, P2, ..., Pn) are also double word aligned. Other areas require word alignment only.



Link area

This area consists of six words, and is at offset zero from the caller's stack pointer on entry to a procedure. The first word contains the caller's back chain (stack pointer). The second word is where the callee saves the Condition Register (CR) if needed. The third word is where the callee saves the Link Register if necessary. The fourth word is reserved for the **SETJMP**, **LONGJMP** processing, and the fifth word is reserved for future use. The last word (word 6) is reserved for use by the Global Linkage routines which are used when calling out–of–module routines (for example, in shared libraries).

Input Parameter Area

This is a contiguous piece of storage reserved by the calling program to represent the register image of the input parameters of the callee. The input parameter area is double word aligned, and is located on the stack directly following the caller's link area. This area is at least 8 words in size. If more than 8 words of parameters are expected, they would have been stored as register images starting at positive offset 56 from the incoming stack pointer.

Register Save area

This area is double word aligned, and provides the space needed to save all non–volatile FPRs and GPRs used by the callee program. The FPRs are saved next to the link area. The GPRs are saved above the FPRs (in lower addresses). The called function may save the registers here even if it does not need to allocate a new stack frame. Locations at a numerically lower address than stack floor should not be accessed.

A callee needs only to save the non–volatile registers that it actually uses. Register 31 is always saved in the highest addressed word of the particular save area.

Local stack area

This is the space allocated by the callee procedure for local variables, and temporaries.

Output Parameter Area

The parameter area (P1...Pn) must be large enough to hold the largest parameter list of all procedures called from the procedure that owns this stack frame.

This area is at least 8 words long regardless of the length or existence of any argument list.

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, you may want to remember the following naming conventions when you create symbol names.

A called routine has two symbols associated with it: a function descriptor (*Name*) and an entry point (*.Name*). When a call is made to a routine, the compiler branches to the name point directly. Excluding the loading of parameters (if any) in the proper registers, calls to functions are expanded by compilers to the following two instruction sequences:

```
bl   .foo              # Branch to foo.
cror 15,15,15          # Special NOP.
```

The linkage editor will do one of two things when it sees the **bl** instruction:

1. If *foo* is imported (not in the same module), then the linkage editor will change the **bl** to *.foo* to a **bl** to *.glink* (global linkage routine) of *foo*, and insert the *.glink* into the module. Also, if a NOP instruction (`cror 15,15,15`) immediately follows the **bl** instruction, the linkage editor will replace the NOP instruction with the **l** (Load) instruction

   `l 2,20(1).`

2. If *foo* is bound in the same module as its caller, and a `l 2,20(1)` instruction immediately follows the **bl** instruction, then it will replace the **l** instruction with a NOP (`cror 15,15,15`).

Note: For any export, the linkage editor will insert the procedure's descriptor into the module.

The Called Routine's Responsibilities

On entry to a routine, some or all of the following steps may have to be done:

1. Save the link register at offset 8 from the stack pointer if necessary.
2. If any of the CR bits 8–19 (CR2, CR3, CR4) are used then save the CR at displacement 4 from the current stack pointer.
3. Save any non–volatile FPRs used by this procedure in the caller's FPR save area. There is a set of routines named **._savef14**, **._savef15**, ... **._savef31** which may be used.
4. Save all non–volatile GPRs used by this procedure in the caller's GPR save area.
5. Store back chain and decrement stack pointer by the size of the stack frame. Note that if a stack overflow occurs, it will be known immediately when the store of the back chain is done.

This sequence of statements is sometimes referred to as the **prolog**.

On exit from a procedure, some or all of the following steps may have to be performed:

1. Restore all GPRs saved.
2. Restore stack pointer to the value it had on entry.
3. Restore link register if necessary.
4. Restore bits 8–19 of the CR if necessary.
5. If any FPRs were saved then restore them using **._restf***n* where *n* is the first FPR to be restored.
6. Return to caller.

This sequence of statements is sometimes referred to as the **epilog**.

Example

The following is an example of assembler code which is called by a C routine:

```
#       Call this assembly routine from C routine:
#        callfile.c:
#        main()
#        {
#        examlinkage();
#        }
#        Compile as follows:
#        cc -o callfile callfile.c examlinkage.s
#
################################################################
#        On entry to a procedure(callee), all or some of the
#        following steps should be done:
#        1.  Save the link register at offset 8 from the
#            stack pointer for non-leaf procedures.
#        2.  If any of the CR bits 8-19(CR2,CR3,CR4) is used
#            then save the CR at displacement 4 of the current
#             stack pointer.
#        3.  Save all non-volatile FPRs used by this routine.
#            If more that three non-volatile FPR are saved,
#             a call to ._savefn can be used to
#            save them (n is the number of the first FPR to be
#             saved).
#        4.  Save all non-volatile GPRs used by this routine
#            in the caller's GPR SAVE area (negative displacement
#            from the current stack pointer r1).
#        5.  Store back chain and decrement stack pointer by the
#            size of the stack frame.
#
#        On exit from a procedure (callee), all or some of the
#        following steps should be done:
#        1.  Restore all GPRs saved.
#        2.  Restore stack pointer to value it had on entry.
#        3.  Restore Link Register if this is a non-leaf procedure.
#        4.  Restore bits 20-31 of the CR is it was saved.
#        5.  Restore all FPRs saved.  If any FPRs were saved then
#            a call to ._savefn can be used to restore them
#            (n is the first FPR to be restored).
#        6.  Return to caller.
################################################################
#        The following routine calls printf() to print a string.
#        The routine performs entry steps 1-5 and exit steps 1-6.
#        The prolog/epilog code is for small stack frame size.
#        DSA + 8 < 32k
################################################################
        .file   "examlinkage.s"
#Static data entry in T(able)O(f)C(ontents)
        .toc
T.examlinkage.c:        .tc     examlinkage.c[tc],examlinkage.c[rw]
        .globl  examlinkage[ds]
#examlinkage[ds] contains definitions needed for
#runtime linkage of function examlinkage
        .csect  examlinkage[ds]
        .long   .examlinkage[PR]
```

```
        .long   TOC[tc0]
        .long   0
#Function entry in T(able)O(f)C(ontents)
        .toc
T.examlinkage:  .tc     .examlinkage[tc],examlinkage[ds]
#Main routine
        .globl  .examlinkage[PR]
        .csect  .examlinkage[PR]
#       Set current routine stack variables
#       These values are specific to the current routine and
#       can vary from routine to routine
        .set    argarea,     32
        .set    linkarea,    24
        .set    locstckarea, 0
        .set    ngprs,       19
        .set    szdsa,
8*nfprs+4*ngprs+linkarea+argarea+locstckarea
#PROLOG:  Called Routines Responsibilities
        #    Get link reg.
        mflr    0
        #    Get CR if current routine alters it.
        mfcr    12
        #    Save FPR's 14-31.
        bl      ._savef14
        cror 0xf, 0xf, 0xf
        #    Save GPR's 13-31.
        stm     13, -8*nfprs-4*ngprs(1)
        #    Save LR if non-leaf routine.
        st      0, 8(1)
        #    Save CR if current routine alters it.
        st      12, 4(1)
        #    Decrement stack ptr and save back chain.
        stu     1, -szdsa(1)
################################
#load static data address
################################
        l       14,T.examlinkage.c(2)
       #  Load string address which is an argument to printf.
              cal 3, printing(14)
        #  Call to printf routine
        bl      .printf[PR]
        cror 0xf, 0xf, 0xf

#EPILOG:  Return Sequence
        #    Restore stack ptr
        ai      1, 1, szdsa
        #    Restore GPR's 13-31.
        lm      13, -8*nfprs-4*ngprs(1)
        #    Restore FPR's 14-31.
        bl      ._restf14
        cror 0xf, 0xf, 0xf
```

```
        #    Get saved LR.
        l       0, 8(1)
        #    Get saved CR if this routine saved it.
        l       12, 4(1)
        #    Move return address to link register.
        mtlr    0
        #    Restore CR2, CR3, & CR4 of the CR.
        mtcrf   0x38,12
        #    Return to address held in Link Register.
        brl
        #       External variables
        .extern ._savef14
        .extern ._restf14
        .extern .printf[PR]
################################
#       Data
################################
        .csect  examlinkage.c[rw]
        .align  2
printing:       .byte   'E,'x,'a,'m,'p,'l,'e,' ,'f,'o,'r,'
                .byte    'P,'R,'I,'N,'T,'I,'N,'G
        .byte   0xa,0x0
```

Understanding the TOC

The TOC, or Table of Contents, of an XCOFF file is analogous to the table of contents of a book. The TOC is used to find objects in an XCOFF file. An XCOFF file is composed of sections that contain different types of data to be used for specific purposes. Some sections can be further subdivided into subsections or *csects*. A csect is the smallest replaceable unit of an XCOFF file. At runtime, the TOC can contain the csect locations (and the locations of labels inside of csects).

The three sections that contain csects are:

1. **.text** – Indicates that this csect contains code or read-only data.
2. **.data** – Indicates that this csect contains read–write data.
3. **.bss** – Indicates that this csect contains unitialized mapped data.

The storage class of the csect determines the section in which the csect is grouped.

The TOC itself is located in the **.data** section of an XCOFF object file, and is composed of TOC entries. A TOC entry is just a csect that happens to contain the address of another csect. When an XCOFF module is loaded, TOC entries are "relocated": the *real* addresses where the csects will reside in memory are filled into each TOC entry. Therefore, to access a csect in the module, two pieces of information are required:

- The location of the beginning of the TOC
- The offset from the beginning of the TOC of the specific TOC entry that points to the csect.

Using the TOC

To use the TOC, you must follow certain conventions.

- General Purpose Register 2 always contains a pointer to the TOC.
- All references from the **.text** section of an assembler program to the **.data** or the **.bss** sections must occur via the TOC.

The TOC register (General Purpose Register 2) is set up by the system when a program is invoked. It must be maintained by any code written. The TOC Register provides module "context" so that any routines in the module can access data items.

The second of these conventions allows the **.text** and **.data** section to be loaded into different locations in memory easily. By following this convention, you can assure that the only part of the module that will need relocating are the TOC entries.

Accessing Data through the TOC

An external data item is easily accessed by first getting that item's address out of the TOC, and then using that address to get the data. In order to do this, proper relocation information must be provided to access the correct TOC entry. Specific assembly language pseudo-ops have been provided that generate the correct information to access a TOC entry. The code in Figure 1 shows how to access an item *a* using its TOC entry.

```
        define(RTOC,2)
        .csect prog1[pr]        #prog1 is a csect
                #containing instrs.
        ...
        l  5,TCA(RTOC)  #Now GPR5 contains the
                #address of a[rw].
        ...
        .toc
TCA:    .tc  a[tc],a[rw]        #1st parameter is TOC entry
                                #name, 2nd is contents of
                                #TOC entry.

        .extern a[rw]   #a[rw] is an external symbol.
```

This same method is used to access a program's "static internal" data. This is all the data that retains its value over a call, but can only be accessed by the procedures in the file where the data items are declared. In C, this is data with the static attribute,

```
static int xyz;
```

This data is given a name that is determined by convention, and in XCOFF it is the name of the data with an underscore in front of it.

```
        .csect prog1[pr]
        ...
        l 1,STprog1(RTOC)       #Load r1 with the address
                                #prog1's static data.
        ...
        .csect _prog1[rw]       #prog1's static data.

        .long    0
        ...
        .toc
STprog1: .tc.prog1[tc],_prog1[rw]       #TOC entry with address of
                                        #prog1's static data.
```

Inter-module Calls Using the TOC

Since all the access from the text to the data section are via the TOC, then another feature of the TOC can be used that allows inter-module calls. This allows routines to be linked together without resolving all the addresses or symbols at linkedit time. In other words, a call can be made to a common utility routine without actually having that routine linked into the same module as the calling routine. In this way groups of routines can be made into

modules, and the routines in the different groups can call each other with the bind time being delayed until load time. However, in order to use this feature, certain conventions must be followed when calling a routine that is in another module.

To call a routine in another module, an interface routine (or *global linkage* routine) is called that will switch context from the current module to the new module. This context switch is easily performed by saving the TOC pointer to the current module, loading the TOC pointer of the new module, and then branching to the new routine in the other module. The other routine then returns to the original , where the original TOC address is loaded into the TOC register.

In order to make global linkage as transparent as possible, a call can be made to an external routine without any knowledge of what module that routine will go into. Figure 3 has an example of a simple call to a routine that may or may not go through global linkage. During bind time, the binder will determine whether to call global linkage code or not, and will "insert" the proper global linkage routine to perform the inter-module call. This is controlled by an IMPORT list. The following example shows calling a routine that may go through global linkage.

```
.csect prog1[pr]
...
.extern prog1[pr]       #prog1 is an external symbol.
bl .prog2[pr]   #Restore TOC address.
cror 15,15,15   #Call the routine the binder may insert.

        #prog2[gl] and have
        #this call go to global
        #linkage code.
```

The following example shows an example of a call through a global linkage routine.

```
        .csect .prog1[pr]
        bl .prog2[GL]   #glue for global linkage
        l 2,stktoc(1)   #Restore TOC address.
        .toc
prog2: .tc prog2[TC],out_of_module[DS]  #toc entry – address of
descriptor
        .extern out_of_module[DS]       #for OUT–OF–MODULE routine.

#RS linkage register conventions:
#       R2  TOC
#       R1  stack pointer
#       LR  return address
        .set stktoc 20

        .csect .prog2[GL]
        .globl .prog2
.prog2:
        st 2,stktoc(1)  #saves callers toc.
        l  12,prog2(2)  #get address of OUT–OF–MODULE descriptor.
        l  2,4(12)      #get his toc.
        l  12,0(12)     #get his entry address.
        mtctr  12       #put in Count Register.
        bctr            #return to entry address (in Count
Register).
```

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:

```
$ progname
```

By default, any program output goes to standard output. To direct output to a place other than the standard output, use the AIX operating system *shell* > operator.

You can diagnose runtime errors by invoking the symbolic debugger with the AIX operating system **dbx** command. This command invokes a symbolic debugger that works with any code that adheres to XCOFF format conventions. You can use **dbx** to debug all compiler and assembler generated code.

Related Information

The **as** Command article in *Commands Reference* explains assembling a program with the **as** command.
The **ld** Command article in *Commands Reference* explains linking the modules of a program with the **ld** command.
The **.csect** pseudo–op, **.toc** pseudo–op, **.tocof** pseudo–op.
The **dbx** command in *General Programming Concepts*.

Chapter 5. Instruction Set

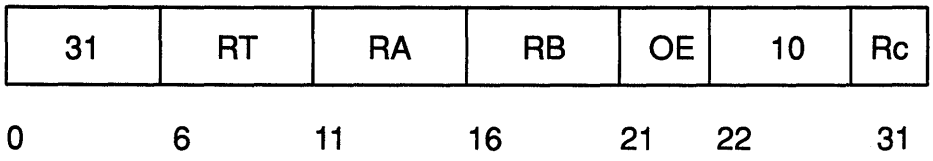
a (Add) Instruction

Purpose

Adds the contents of two general purpose registers and places the result in a general purpose register.

Syntax

a	*RT,RA,RB*
a.	*RT,RA,RB*
ao	*RT,RA,RB*
ao.	*RT,RA,RB*



Description

The **a** instruction places the sum of the contents of General Purpose Register *RA* and General Purpose Register *RB* into the target General Purpose Register *RT*.

The **a** instruction has four syntax forms. Each syntax form has a different effect on Condition Register Field 0 and the Fixed Point Exception Register.

Syntax form	Overflow Exception (OE)	Fixed Point Exception Register	Record bit (Rc)	Condition Register Field 0
a	0	CA	0	None
a.	0	CA	1	LT,GT,EQ,SO
ao	1	SO,OV,CA	0	None
ao.	1	SO,OV,CA	1	LT,GT,EQ,SO

The four syntax forms of the **a** instruction always affect the Carry bit (CA) in the Fixed Point Exception Register. If the syntax form sets the Overflow Exception (OE) bit to 1, the instruction affects the Summary Overflow (SO) and Overflow (OV) bits in the Fixed Point Exception Register. If the syntax form sets the Record (Rc) bit to 1, the instruction affects the Less Than (LT) zero, Greater Than (GT) zero, Equal To (EQ) zero, and Summary Overflow (SO) bits in Condition Register Field 0.

Parameters

RT	Specifies target general purpose register where result of operation is stored.
RA	Specifies source general purpose register for operation.
RB	Specifies source general purpose register for operation.

Examples

1. To add the contents of GPR 4 to the contents of GPR 10 and store the result in GPR 6:

```
# Assume GPR 4 contains 0x9000 3000.
# Assume GPR 10 contains 0x8000 7000.
a 6,4,10
# GPR 6 now contains 0x1000 A000.
```

2. To add the contents of GPR 4 to the contents of GPR 10, store the result in GPR 6, and set Condition Register Field 0 to reflect the result of the operation:

```
# Assume GPR 4 contains 0x7000 3000.
# Assume GPR 10 contains 0xFFFF FFFF.
a. 6,4,10
# GPR 6 now contains 0x7000 2FFF.
```

3. To add the contents of GPR 4 to the contents of GPR 10, store the result in GPR 6, and set the Summary Overflow, Overflow, and Carry bits in the Fixed Point Exception Register to reflect the result of the operation:

```
# Assume GPR 4 contains 0x9000 3000.
# Assume GPR 10 contains 0x7B41 92C0.
ao 6,4,10
# GPR 6 now contains 0x0B41 C2C0.
```

4. To add the contents of GPR 4 to the contents of GPR 10, store the result in GPR 6, and set the Summary Overflow, Overflow, and Carry bits in the Fixed Point Exception Register and Condition Register Field 0 to reflect the result of the operation:

```
# Assume GPR 4 contains 0x8000 0000.
# Assume GPR 10 contains 0x8000 7000.
ao. 6,4,10
# GPR 6 now contains 0x0000 7000.
```

Implementation Specifics

This instruction is part of Application Development Toolkit in AIX Base.

Related Information

Understanding Fixed Point Arithmetic Instructions on page 1–8.

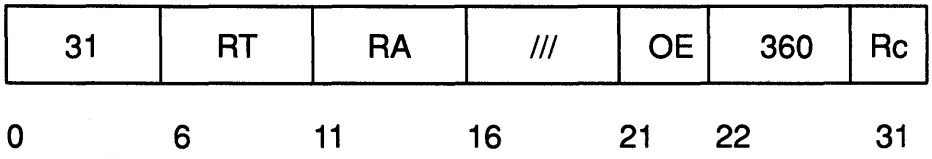
abs (Absolute) Instruction

Purpose

Takes the absolute value of the contents of a general purpose register and places the result in a general purpose register.

Syntax

abs *RT,RA*
abs. *RT,RA*
abso *RT,RA*
abso. *RT,RA*



Description

The **abs** instruction places the absolute value of the contents of General Purpose Register *RA* into the target General Purpose Register *RT*.

If General Purpose Register *RA* contains the most negative number ('8000 0000'), the result of the instruction is the most negative number, and the instruction will set the Overflow bit in the Fixed Point Exception Register to 1 if the OE bit is set to 1.

The **abs** instruction has four syntax forms. Each syntax form has a different effect on Condition Register Field 0 and the Fixed Point Exception Register.

Syntax form	Overflow Exception (OE)	Fixed Point Exception Register	Record bit (Rc)	Condition Register Field 0
abs	0	None	0	None
abs.	0	None	1	LT,GT,EQ,SO
abso	1	SO,OV	0	None
abso.	1	SO,OV	1	LT,GT,EQ,SO

The four syntax forms of the **abs** instruction always affect the Carry bit (CA) in the Fixed Point Exception Register. If the syntax form sets the Overflow Exception (OE) bit to 1, the instruction affects the Summary Overflow (SO) and Overflow (OV) bits in the Fixed Point Exception Register. If the syntax form sets the Record (Rc) bit to 1, the instruction affects the Less Than (LT) zero, Greater Than (GT) zero, Equal To (EQ) zero, and Summary Overflow (SO) bits in Condition Register Field 0.

Parameters

RT Specifies target general purpose register where result of operation is stored.

RA Specifies source general purpose register for operation.

Examples

1. To take the absolute value of the contents of GPR 4 and store the result in GPR 6:

```
# Assume GPR 4 contains 0x7000 3000.
abs 6,4
# GPR 6 now contains 0x7000 3000.
```

2. To take the absolute value of the contents of GPR 4, store the result in GPR 6, and set Condition Register Field 0 to reflect the result of the operation:

```
# Assume GPR 4 contains 0xFFFF FFFF.
abs. 6,4
# GPR 6 now contains 0x0000 0001.
```

3. To take the absolute value of the contents of GPR 4, store the result in GPR 6, and set the Summary Overflow and Overflow bits in the Fixed Point Exception Register to reflect the result of the operation:

```
# Assume GPR 4 contains 0xB004 3000.
abso 6,4
# GPR 6 now contains 0x4FFB D000.
```

4. To take the absolute value of the contents of GPR 4, store the result in GPR 6, and set the Summary Overflow and Overflow bits in the Fixed Point Exception Register and Condition Register Field 0 to reflect the result of the operation:

```
# Assume GPR 4 contains 0x8000 0000.
abso. 6,4
# GPR 6 now contains 0x8000 0000.
```

Implementation Specifics

This instruction is part of Application Development Toolkit in AIX Base.

Related Information

Understanding Fixed Point Arithmetic Instructions on page 1–8.

ae (Add Extended) Instruction

Purpose

Adds the contents of two general purpose registers to the value of the Carry bit in the Fixed Point Exception Register and places the result in a general purpose register.

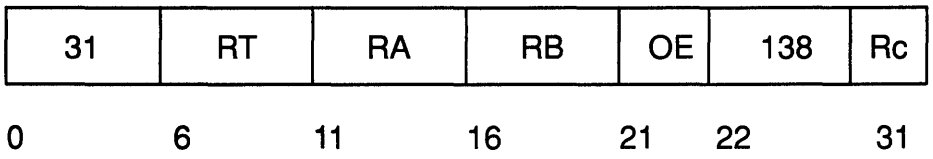
Syntax

ae *RT,RA,RB*

ae. *RT,RA,RB*

aeo *RT,RA,RB*

aeo. *RT,RA,RB*



Description

The **ae** instruction places sum of the contents of General Purpose Register *RA*, General Purpose Register *RB*, and the Carry bit into the target General Purpose Register *RT*.

The **ae** instruction has four syntax forms. Each syntax form has a different effect on Condition Register Field 0 and the Fixed Point Exception Register.

Syntax form	Overflow Exception (OE)	Fixed Point Exception Register	Record bit (Rc)	Condition Register Field 0
ae	0	CA	0	None
ae.	0	CA	1	LT,GT,EQ,SO
aeo	1	SO,OV,CA	0	None
aeo.	1	SO,OV,CA	1	LT,GT,EQ,SO

The four syntax forms of the **ae** instruction always affect the Carry bit (CA) in the Fixed Point Exception Register. If the syntax form sets the Overflow Exception (OE) bit to 1, the instruction affects the Summary Overflow (SO) and Overflow (OV) bits in the Fixed Point Exception Register. If the syntax form sets the Record (Rc) bit to 1, the instruction affects the Less Than (LT) zero, Greater Than (GT) zero, Equal To (EQ) zero, and Summary Overflow (SO) bits in Condition Register Field 0.

Parameters

RT Specifies target general purpose register where result of operation is stored.

RA Specifies source general purpose register for operation.

RB Specifies source general purpose register for operation.

Examples

1. To add the contents of GPR 4, the contents of GPR 10, and the Fixed Point Exception Register Carry bit and store the result in GPR 6:

```
# Assume GPR 4 contains 0x1000 0400.
# Assume GPR 10 contains 0x1000 0400.
# Assume the Carry bit is one.
ae 6,4,10
# GPR 6 now contains 0x2000 0801.
```

2. To add the contents of GPR 4, the contents of GPR 10, and the Fixed Point Exception Register Carry bit, store the result in GPR 6, and to set Condition Register Field 0 to reflect the result of the operation:

```
# Assume GPR 4 contains 0x9000 3000.
# Assume GPR 10 contains 0x7B41 92C0.
# Assume the Carry bit is zero.
ae. 6,4,10
# GPR 6 now contains 0x0B41 C2C0.
```

3. To add the contents of GPR 4, the contents of GPR 10, and the Fixed Point Exception Register Carry bit, store the result in GPR 6, and to set the Summary Overflow, Overflow, and Carry bits in the Fixed Point Exception Register to reflect the result of the operation:

```
# Assume GPR 4 contains 0x1000 0400.
# Assume GPR 10 contains 0xEFFF FFFF.
# Assume the Carry bit is one.
aeo 6,4,10
# GPR 6 now contains 0x0000 0400.
```

4. To add the contents of GPR 4, the contents of GPR 10, and the Fixed Point Exception Register Carry bit, store the result in GPR 6, and set the Summary Overflow, Overflow, and Carry bits in the Fixed Point Exception Register and Condition Register Field 0 to reflect the result of the operation:

```
# Assume GPR 4 contains 0x9000 3000.
# Assume GPR 10 contains 0x8000 7000.
# Assume the Carry bit is zero.
aeo. 6,4,10
# GPR 6 now contains 0x1000 A000.
```

Implementation Specifics

This instruction is part of Application Development Toolkit in AIX Base.

Related Information

Understanding Fixed Point Arithmetic Instructions on page 1–8.

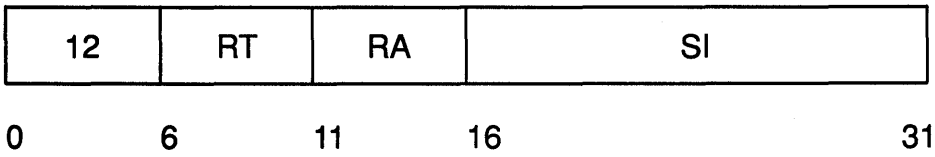
ai (Add Immediate) Instruction

Purpose

Adds the contents of a general purpose register and a 16-bit signed integer, places the result in a general purpose register, and effects the Carry bit of the Fixed Point Exception Register.

Syntax

ai *RT,RA,SI*



Description

The **ai** instruction places the sum of the contents of General Purpose Register *RA* and a 16-bit signed integer *SI* into target General Purpose Register *RT*.

The 16-bit integer provided as immediate data is sign-extended to 32 bits prior to carrying out the addition operation.

The **ai** instruction has one syntax form and can set the Carry bit of the Fixed Point Exception Register; it never affects Condition Register Field 0.

Parameters

RT	Specifies target general purpose register where result of operation is stored.
RA	Specifies source general purpose register for operation.
SI	Specifies 16–bit signed integer for operation.

Examples

1. To add 0xFFFF FFFF to the contents of GPR 4, store the result in GPR 6, and set the Carry bit to reflect the result of the operation:

```
# Assume GPR 4 contains 0x0000 2346.
ai 6,4,0xFFFFFFFF
# GPR 6 now contains 0x0000 2345.
```

Implementation Specifics

This instruction is part of Application Development Toolkit in AIX Base.

Related Information

Understanding Fixed Point Arithmetic Instructions on page 1–8.

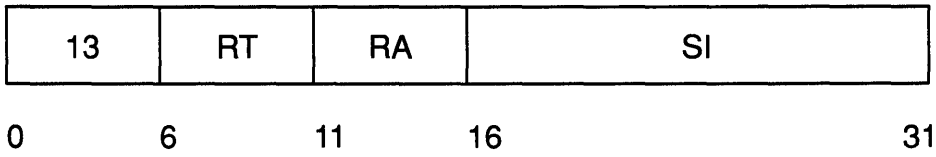
ai. (Add Immediate and Record) Instruction

Purpose

Adds the contents of a general purpose register and a 16-bit signed integer, places the result in another general purpose register, and affects the Carry bit of the Fixed Point Exception Register and Condition Register Field 0.

Syntax

ai. *RT,RA,SI*



Description

The **ai.** instruction places the sum of the contents of General Purpose Register *RA* and a 16-bit signed integer *SI* into the target General Purpose Register *RT*.

The 16-bit integer *SI* provided as immediate data is sign-extended to 32 bits prior to carrying out the addition operation.

The **ai.** instruction has one syntax form and can set the Carry Bit of the Fixed Point Exception Register. This instruction also affects Condition Register Field 0.

Parameters

RT	Specifies target general purpose register where result of operation is stored.
RA	Specifies source general purpose register for operation.
SI	Specifies 16–bit signed integer for operation.

Examples

1. To add a 16-bit signed integer to the contents of GPR 4 , store the result in GPR 6, and set the Fixed Point Exception Register Carry bit and Condition Register Field 0 to reflect the result of the operation:

```
# Assume GPR 4 contains 0xEFFF FFFF.
ai. 6,4,0x1000
# GPR 6 now contains 0xF000 0FFF.
```

Implementation Specifics

This instruction is part of Application Development Toolkit in AIX Base.

Related Information

Understanding Fixed Point Arithmetic Instructions on page 1–8.

ame (Add to Minus One Extended) Instruction

Purpose

Adds the contents of a general purpose register, the Carry bit in the Fixed Point Exception Register, and –1 and places the result in a general purpose register.

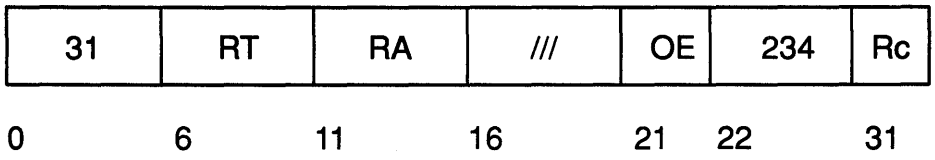
Syntax

ame *RT,RA*

ame. *RT,RA*

ameo *RT,RA*

ameo. *RT,RA*



Description

The **ame** instruction places the sum of the contents of General Purpose Register *RA*, the Carry bit of the Fixed Point Exception Register, and –1 (0xFFFF FFFF) into the target General Purpose Register *RT*.

The **ame** instruction has four syntax forms. Each syntax form has a different effect on Condition Register Field 0 and the Fixed Point Exception Register.

Syntax form	Overflow Exception (OE)	Fixed Point Exception Register	Record bit (Rc)	Condition Register Field 0
ame	0	CA	0	None
ame.	0	CA	1	LT,GT,EQ,SO
ameo	1	SO,OV,CA	0	None
ameo.	1	SO,OV,CA	1	LT,GT,EQ,SO

The four syntax forms of the **ame** instruction always affect the Carry bit (CA) in the Fixed Point Exception Register. If the syntax form sets the Overflow Exception (OE) bit to 1, the instruction affects the Summary Overflow (SO) and Overflow (OV) bits in the Fixed Point Exception Register. If the syntax form sets the Record (Rc) bit to 1, the instruction affects the Less Than (LT) zero, Greater Than (GT) zero, Equal To (EQ) zero, and Summary Overflow (SO) bits in Condition Register Field 0.

Parameters

RT Specifies target general purpose register where result of operation is stored.

RA Specifies source general purpose register for operation.

Examples

1. To add the contents of GPR 4, the Carry bit in the Fixed Point Exception Register, and –1 and store the result in GPR 6:

```
# Assume GPR 4 contains 0x9000 3000.
# Assume the Carry bit is zero.
ame 6,4
# GPR 6 now contains 0x9000 2FFF.
```

2. To add the contents of GPR 4, the Carry bit in the Fixed Point Exception Register, and –1, store the result in GPR 6, and set Condition Register Field 0 to reflect the result of the operation:

```
# Assume GPR 4 contains 0xB000 42FF.
# Assume the Carry bit is zero.
ame. 6,4
# GPR 6 now contains 0xB000 42FE.
```

3. To add the contents of GPR 4, the Carry bit in the Fixed Point Exception Register, and –1, store the result in GPR 6, and set the the Summary Overflow, Overflow, and Carry bits in the Fixed Point Exception Register to reflect the result of the operation:

```
# Assume GPR 4 contains 0x8000 0000.
# Assume the Carry bit is zero.
ameo 6,4
# GPR 6 now contains 0x7FFF FFFF.
```

4. To add the contents of GPR 4, the Carry bit in the Fixed Point Exception Register, and –1, store the result in GPR 6, and set the Summary Overflow, Overflow, and Carry bits in the Fixed Point Exception Register and Condition Register Field 0 to reflect the result of the operation:

```
# Assume GPR 4 contains 0x8000 0000.
# Assume the Carry bit is one.
ameo. 6,4
# GPR 6 now contains 0x8000 000.
```

Implementation Specifics

This instruction is part of Application Development Toolkit in AIX Base.

Related Information

Understanding Fixed Point Arithmetic Instructions on page 1–8.

and (AND) Instruction

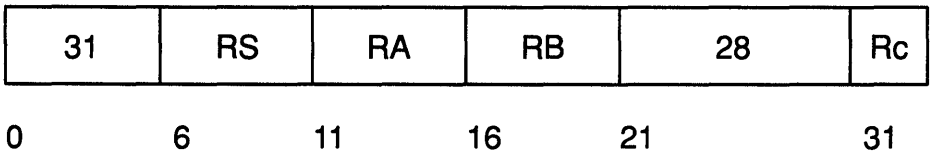
Purpose

Logically ANDs the contents of two general purpose registers and places the result in a general purpose register.

Syntax

and *RA,RS,RB*

and. *RA,RS,RB*



Description

The **and** instruction logically ANDs the contents of General Purpose Register *RS* with the contents of General Purpose Register *RB* and places the result into the target General Purpose Register *RA*.

The **and** instruction has two syntax forms. Each syntax form has a different effect on Condition Register Field 0.

Syntax form	Overflow Exception (OE)	Fixed Point Exception Register	Record bit (Rc)	Condition Register Field 0
and	None	None	0	None
and.	None	None	1	LT,GT,EQ,SO

The two syntax forms of the **and** instruction never affect the Fixed Point Exception Register. If the syntax form sets the Record (Rc) bit to 1, the instruction affects the Less Than (LT) zero, Greater Than (GT) zero, Equal To (EQ) zero, and Summary Overflow (SO) bits in Condition Register Field 0.

Parameters

RA Specifies target general purpose register where result of operation is stored.

RS Specifies source general purpose register for operation.

RB Specifies source general purpose register for operation.

Examples

1. To logically AND the contents of GPR 4 with the contents of GPR 7 and store the result in GPR 6:

```
# Assume GPR 4 contains 0xFFF2 5730.
# Assume GPR 7 contains 0x7B41 92C0.
and 6,4,7
# GPR 6 now contains 0x7B40 1200.
```

2. To logically AND the contents of GPR 4 with the contents of GPR 7, store the result in GPR 6, and set Condition Register Field 0 to reflect the result of the operation:

```
# Assume GPR 4 contains 0xFFF2 5730.
# Assume GPR 7 contains 0xFFFF EFFF.
and. 6,4,7
# GPR 6 now contains 0xFFF2 4730.
```

Implementation Specifics

This instruction is part of Application Development Toolkit in AIX Base.

Related Information

Understanding Fixed Point Logical Instructions on page 1–9.

andc (AND With Complement) Instruction

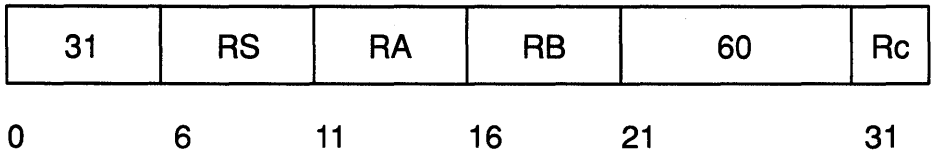
Purpose

ANDs the contents of a general purpose register with the complement of the contents of a general purpose register.

Syntax

andc *RA,RS,RB*

andc. *RA,RS,RB*



Description

The **andc** instruction ANDs the contents of General Purpose Register *RS* with the complement of the contents of General Purpose Register *RB* and places the result into General Purpose Register *RA*.

The **andc** instruction has two syntax forms. Each syntax form has a different effect on Condition Register Field 0.

Syntax form	Overflow Exception (OE)	Fixed Point Exception Register	Record bit (Rc)	Condition Register Field 0
andc	None	None	0	None
andc.	None	None	1	LT,GT,EQ,SO

The two syntax forms of the **andc** instruction never affect the Fixed Point Exception Register. If the syntax form sets the Record (Rc) bit to 1, the instruction affects the Less Than (LT) zero, Greater Than (GT) zero, Equal To (EQ) zero, and Summary Overflow (SO) bits in Condition Register Field 0.

Parameters

RA Specifies target general purpose register where result of operation is stored.

RS Specifies source general purpose register for operation.

RB Specifies source general purpose register for operation.

Examples

1. To AND the contents of GPR 4 with the complement of the contents of GPR 5 and store the result in GPR 6:

```
# Assume GPR 4 contains 0x9000 3000.
# Assume GPR 5 contains 0xFFFF FFFF.
# The complement of 0xFFFF FFFF becomes 0x0000 0000.
andc 6,4,5
# GPR 6 now contains 0x0000 0000.
```

2. To AND the contents of GPR 4 with the complement of the contents of GPR 5, store the result in GPR 6, and set Condition Register Field 0 to reflect the result of the operation:

```
# Assume GPR 4 contains 0xB004 3000.
# Assume GPR 5 contains 0x7676 7676.
# The complement of 0x7676 7676 is 0x8989 8989.
andc. 6,4,5
# GPR 6 now contains 0x8000 0000.
```

Implementation Specifics

This instruction is part of Application Development Toolkit in AIX Base.

Related Information

Understanding Fixed Point Logical Instructions on page 1–9.

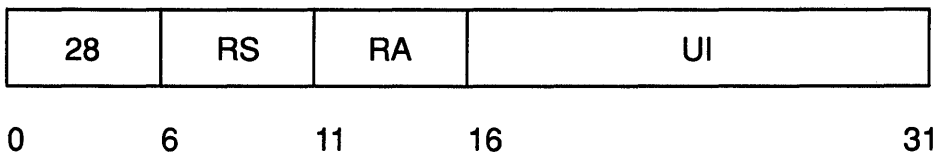
andil. (AND Immediate Lower) Instruction

Purpose

AND's the least significant 16 bits of the contents of a general purpose register with a 16-bit unsigned integer and stores the result in a general purpose register..

Syntax

andil. *RA,RS,UI*



Description

The **andil.** instruction ANDs the contents of General Purpose Register *RS* with the concatenation of x'0000' and a 16-bit unsigned integer *UI* and places the result in General Purpose Register *RA*.

The **andil.** instruction has one syntax form and never affects the Fixed Point Exception Register. This instruction sets the Less Than (LT) zero, Greater Than (GT) zero, Equal To (EQ) zero, or Summary Overflow (SO) bit in Condition Register Field 0.

Parameters

RA	Specifies target general purpose register where result of operation is stored.
RS	Specifies source general purpose register for operation.
UI	Specifies 16–bit unsigned integer for operation.

Examples

1. To AND the contents of GPR 4 with 0x0000 5730, store the result in GPR 6, and set Condition Register Field 0 to reflect the result of the operation:

```
# Assume GPR 4 contains 0x7B41 92C0.
andil. 6,4,0x5730
# GPR 6 now contains 0x0000 1200.
# CRF 0 now contains 0x4.
```

Implementation Specifics

This instruction is part of Application Development Toolkit in AIX Base.

Related Information

Understanding Fixed Point Logical Instructions on page 1–9.

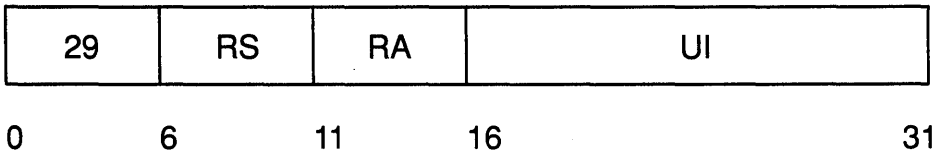
andiu. (AND Immediate Upper) Instruction

Purpose

AND's the most significant 16 bits of the contents of a general purpose register with a 16-bit unsigned integer and stores the result in a general purpose register.

Syntax

andiu. *RA,RS,UI*



Description

The **andiu.** instruction ANDs the contents of General Purpose Register *RS* with the concatenation of a 16-bit unsigned integer *UI* and x'0000' and places the result into the target General Purpose Register *RA*.

The **andiu.** instruction has one syntax form and never affects the Fixed Point Exception Register. This instruction sets the Less Than (LT) zero, Greater Than (GT) zero, Equal To (EQ) zero, or Summary Overflow (SO) bit in Condition Register Field 0.

Parameters

RA	Specifies target general purpose register where result of operation is stored.
RS	Specifies source general purpose register for operation.
UI	Specifies 16–bit unsigned integer for operation.

Examples

1. To AND the contents of GPR 4 with 0x5730 0000, store the result in GPR 6, and set Condition Register Field 0 to reflect the result of the operation:

```
# Assume GPR 4 contains 0x7B41 92C0.
andiu. 6,4,0x5730
# GPR 6 now contains 0x5300 0000.
```

Implementation Specifics

This instruction is part of Application Development Toolkit in AIX Base.

Related Information

Understanding Fixed Point Logical Instructions on page 1–9.

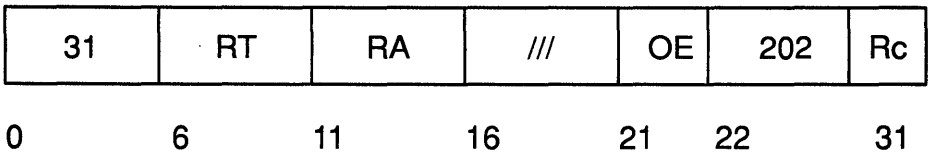
aze (Add To Zero Extended) Instruction

Purpose

Adds the contents of a general purpose register, zero, and the value of the Carry bit in the FIxed Point Exception Register and places the result in a general purpose register.

Syntax

aze *RT,RA*
aze. *RT,RA*
azeo *RT,RA*
azeo. *RT,RA*



Description

The **aze** instruction adds the contents of General Purpose Register *RA*, the Carry bit, and 0x0000 0000 and places the result into the target General Purpose Register *RT*.

The **aze** instruction has four syntax forms. Each syntax form has a different effect on Condition Register Field 0 and the Fixed Point Exception Register.

Syntax form	Overflow Exception (OE)	Fixed Point Exception Register	Record bit (Rc)	Condition Register Field 0
aze	0	CA	0	None
aze.	0	CA	1	LT,GT,EQ,SO
azeo	1	SO,OV,CA	0	None
azeo.	1	SO,OV,CA	1	LT,GT,EQ,SO

The four syntax forms of the **aze** instruction always affect the Carry bit (CA) in the Fixed Point Exception Register. If the syntax form sets the Overflow Exception (OE) bit to 1, the instruction affects the Summary Overflow (SO) and Overflow (OV) bits in the Fixed Point Exception Register. If the syntax form sets the Record (Rc) bit to 1, the instruction affects the Less Than (LT) zero, Greater Than (GT) zero, Equal To (EQ) zero, and Summary Overflow (SO) bits in Condition Register Field 0.

Parameters

RT Specifies target general purpose register where result of operation is stored.

RA Specifies source general purpose register for operation.

Examples

1. To add the contents of GPR 4, zero, and the Carry bit and store the result in GPR 6:

```
# Assume GPR 4 contains 0x7B41 92C0.
# Assume the Carry bit is zero.
aze 6,4
# GPR 6 now contains 0x7B41 92C0.
```

2. To add the contents of GPR 4, zero, and the Carry bit, store the result in GPR 6, and set Condition Register Field 0 to reflect the result of the operation:

```
# Assume GPR 4 contains 0xEFFF FFFF.
# Assume the Carry bit is one.
aze. 6,4
# GPR 6 now contains 0xF000 0000.
```

3. To add the contents of GPR 4, zero, and the Carry bit, store the result in GPR 6, and set the Summary Overflow, Overflow, and Carry bits in the Fixed Point Exception Register to reflect the result of the operation:

```
# Assume GPR 4 contains 0x9000 3000.
# Assume the Carry bit is one.
azeo 6,4
# GPR 6 now contains 0x9000 3001.
```

4. To add the contents of GPR 4, zero, and the Carry bit, store the result in GPR 6, and set the Summary Overflow, Overflow, and Carry bits in the Fixed Point Exception Register and Condition Register Field 0 to reflect the result of the operation:

```
# Assume GPR 4 contains 0xEFFF FFFF.
# Assume the Carry bit is zero.
azeo. 6,4
# GPR 6 now contains 0xEFFF FFFF.
```

Implementation Specifics

This instruction is part of Application Development Toolkit in AIX Base.

Related Information

Understanding Fixed Point Arithmetic Instructions on page 1–8.

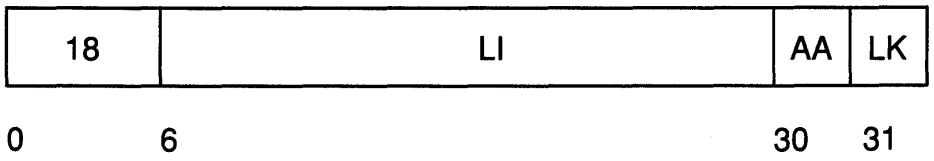
b (Branch) Instruction

Purpose

Branches to a specified target address.

Syntax

b	*target_address*
ba	*target_address*
bl	*target_address*
bla	*target_address*



Description

The **b** instruction branches to an instruction specified by the branch target address. The branch target address is computed one of two ways.

- If the absolute address bit (AA) is 0, then the branch target address is computed by concatenating the 24-bit *LI* field, which is calculated by subtracting the address of the instruction from the target address and dividing the result by four, and b'00', sign-extending the result to 32 bits, and adding this to the address of this branch instruction.
- If the Absolute Address is 1, then the branch target address is LI concatenated with b'00' sign-extended to 32 bits. The *LI* field is the low–order 26 bits of the target address divided by four.

The **b** instruction has four syntax forms. Each syntax form has a different effect on the Link bit and Link Register.

Syntax form	Absolute Address bit (AA)	Fixed Point Exception Register	Link bit (LK)	Condition Register Field 0
b	0	None	0	None
ba	1	None	0	None
bl	0	None	1	None
bla	1	None	1	None

The four syntax forms of the **b** instruction never affect the Fixed Point Exception Register or Condition Register Field 0. The syntax forms set the absolute address (AA) bit and the Link bit (LK) and determine which method of calculating the branch target address is used. If the Link Bit (LK) is set to 1, then the effective address of the instruction is placed in the Link Register.

Parameters

target_address Specifies the target address.

Examples

1. To transfer the execution of the program to *there*:

```
here: b there
      cror 15,15,15
# The execution of the program continues at there.
there:
```

2. To transfer the execution of the program to and set the Link Register:

```
        bl here
return: cror 15,15,15
# The Link Register now contains the address of return.
# The execution of the program continues at here.
```

Implementation Specifics

This instruction is part of Application Development Toolkit in AIX Base.

Related Information

Understanding Branch Instructions on page 1–3.

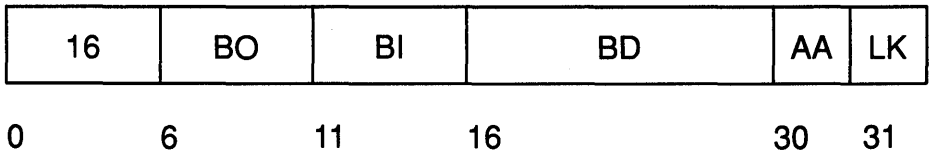
bb (Branch on Condition Register Bit) Instruction

Purpose

Branches to a specified address depending on the value of a specified Condition Register bit.

Syntax

bbt *I1,A2*
bbtl *I1,A2*
bbf *I1,A2*
bbfl *I1,A2*



Description

The **bb** instruction branches to an instruction specified by the branch target address depending on the value of a specified Condition Register bit.

- Use the **bbt** instruction to branch if the specified Condition Register bit is *true* (BD=0xC).
- Use the **bbf** instruction to branch if the specified Condition Register bit is *false* (BD=0x4).

For the **bb** extended mnemonic forms, the assembler subtracts the address of the branch instruction from the address *A2*. It then divides the result by 4 to get the branch displacement (BD) in full words. At runtime, bit *I1* of the Condition Register is checked. *I1* must be greater than or equal to 0 and less than or equal to 31. If the condition is satisfied, the concatenation of the branch displacement and b'00' is sign-extended and the result is added to the address of the branch instruction. The result is the branch target address.

Syntax form	Branches if CR bit is:	Fixed Point Exception Register	Link bit (LK)	Condition Register Field 0
bbt	True	None	0	None
bbtl	True	None	1	None
bbf	Flase	None	0	None
bbfl	False	None	1	None

The four syntax forms of the **bb** instruction never affect the Fixed Point Exception Register or Condition Register Field 0. If the Link Bit (LK) is set to 1, then the effective address of the instruction is placed in the Link Register.

Parameters

I1 Specifies bit in Condition Register for condition comparison.

A2 Specifies address used in calculation of branch displacement.

Extended Mnemonics

Three extended mnemonic branch instructions are based on the various branch instructions. The extended mnemonic **a** form is based on the **bb** (Branch on Condition Register bit) instruction. The extended mnemonic **c** form is based on the **bcc** (Branch Conditional to Count Register) instruction. The extended mnemonic **r** form is based on the **bcr** (Branch Conditional Register) instruction. These instructions check bit *I1* of the Condition Register for condition evaluation.

- The extended mnemonic **a** defines the branch target address as BD||'00'.
- The extended mnemonic **c** form defines the branch target address as the contents of the Count Register.
- The extended mnemonic **r** form defines the branch target address as the contents of the Link Register.
- The **t** and **f** in the mnemonic instructions represent **TRUE** and **FALSE** for the branch conditions.

Use the **l** form to place the instruction following the Branch Instruction in the Link Register.

Syntax	Parameters	Description
bbta, bbfa,	*I1,A2*	Branch on CR bit
bbtla, bbfla	*I1,I2*	
bbtc, bbfc,	*I1*	Branch Count Register
bbtcl, bbfcl	*I1*	on CR Bit
bbtr, bbfr,	*I1*	Branch Register on CR Bit
bbtrl, bbfrl	*I1*	

Examples

1. To branch to a new address dependant on the third bit of the Condition Register:

```
here: si 6,5,0x1
# Assume GPR 5 equals 1.
# One is subtracted from GPR 5 and the result is stored
# in GPR 6.
cmpi 0,6,0x0
# GPR 6 is compared to zero and the result is recorded
# in the first four bits of the Condition Register.
bbt 2,here
# The branch to here occurs if the comparison is equal.
```

2. To branch to a new address dependant on the third bit of the Condition Register and place the address following the branch in the Link Register.

```
there: ai 6,5,-1
# Assume GPR 5 equals 5.
# -1 is added to GPR 5 and the result is stored
# in GPR 6.
cmpi 0,6,0x0
# GPR 1 is compared to zero and the result is recorded
# in the first four bits of the Condition Register.
bbfl 2,there
# The branch to there occurs if the comparison
# is not equal.
```

Implementation Specifics

This instruction is part of Application Development Toolkit in AIX Base.

Related Information

Understanding Branch Instructions on page 1–3.

bc (Branch Conditional) Instruction

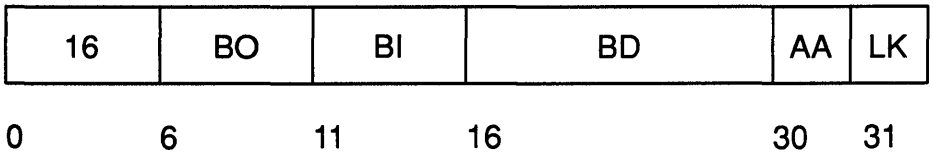
Purpose

Conditionally branches to a specified target address.

Syntax

bc	*BO,BI,target_address*
bca	*BO,BI,target_address*
bcl	*BO,BI,target_address*
bcla	*BO,BI,target_address*

Extended mnemonics are also provided.



Description

The **bc** instruction branches to an instruction specified by the branch target address. The branch target address is computed one of two ways.

- If the absolute address bit (AA) is 0, then the branch target address is computed by concatenating the 24-bit Branch Displacement (BD) and b'00', sign-extending this to 32 bits, and adding the result to the address of this branch instruction.
- If the Absolute Address is 1, then the branch target address is BD concatenated with b'00' sign-extended to 32 bits.

The **bc** instruction has four syntax forms. Each syntax form has a different effect on Condition Register Field 0 and the Fixed Point Exception Register.

Syntax form	Absolute Address bit (AA)	Fixed Point Exception Register	Link bit (LK)	Condition Register Field 0
bc	0	None	0	None
bca	1	None	0	None
bcl	0	None	1	None
bcla	1	None	1	None

The four syntax forms of the **bc** instruction never affect the Fixed Point Exception Register or Condition Register Field 0. The syntax forms set the absolute address (AA) bit and the Link bit (LK) and determine which method of calculating the branch target address is used. If the Link Bit (LK) is set to 1, then the effective address of the instruction is placed in the Link Register.

The Branch Option field (BO) is used to combine different types of branches into a single instruction. Extended mnemonics are provided to set the Branch Option field automatically. The Branch Option field has one of the following specifications, where *x* stands for either a 0 or a 1:

BO	Description
0000x	Decrement the Count Register, then branch if the decremented CTR is not 0 and condition FALSE.
0001x	Decrement the Count Register, then branch if the decremented CTR is 0 and condition FALSE.
001xx	Branch if condition FALSE.
0100x	Decrement the Count Register, then branch if the decremented CTR is not 0 and condition TRUE.
0101x	Decrement the Count Register, then branch if the decremented CTR is 0 and condition TRUE.
011xx	Branch if condition TRUE.
1x00x	Decrement the Count Register, then branch if the decremented CTR is not 0.
1x01x	Decrement the Count Register, then branch if the decremented CTR is 0.
1x1xx	Branch always.

Parameters

target_address	Specifies the target address. For absolute branches such as **bca** and **bcla**, the target address can be immediate data containable in 16 bits.
BI	Specifies bit in Condition Register for condition comparison.
BO	Specifies Branch Option field used in instruction.
BIF	Specifies the Condition Register field that specifies the Condition Register bit (LT, GT, EQ, SO) to be used for condition comparison.

Extended Mnemonics

Six extended mnemonic branch commands are based on the **bc** command. In the branch and decrement commands that begin with **bd**, use the **bdz** form to branch if CTR equals 0 and the **bdn** form to branch if CTR does not equal zero. Use the **l** form to place the instruction that follows the Branch Instruction in the Link Register, and use the 12 Branch Codes to replace **XX** in each Extended Mnemonic command.

Syntax	Parameters	Description
bXX, bXXl	[*BIF*],*target_address*	Branch on Condition Extended
bXXa, bXXla	[*BIF*],*target_address*	Branch on Condition Extended Absolute
bdz, bdn, bdzl, bdnl	*target_address*	Branch and Decrement CTR
bdza, bdna, bdzla, bdnla	*target_address*	Branch Absolute and Decrement CTR

bdzr, bdnr, bdzrl, bdnrl	None	Branch Register and Decrement CTR
bdzXX, bdnXX	*target_address*	Branch and Decrement CTR on Condition*

**This command uses only the first eight Branch Codes (also marked by *) in place of XX.*

Examples

1. To branch to a target address dependent on the value in the Count Register:

```
lil 8,0x3
# Loads GPR 2 with 0x3.
mtctr 8
# The Count Register equals 0x3.
ai. 9,8,0x1
# Adds one to GPR 8 and places the result in GPR 9.
# The Condition Register records a comparison against zero
# with the result.
bc 0xC,0,there
# Branch is taken if condition is true. 0 indicates that
# the 0 bit in the Condition Register is checked to
# determine if it is set (the LT bit is on). If it is set,
# back occurs.
bcl 0x8,2,there
# Count Register is decremented by one, and CTR becomes 2.
# The branch occurs if CTR is not equal to 0 and Condition
# Register bit 2 is set (the EQ bit is on).
# The Link Register contains address of next instruction.
```

Implementation Specifics

This instruction is part of Application Development Toolkit in AIX Base.

Related Information

Understanding Branch Instructions on page 1–3.

bcc (Branch Conditional to Count Register) Instruction

Purpose

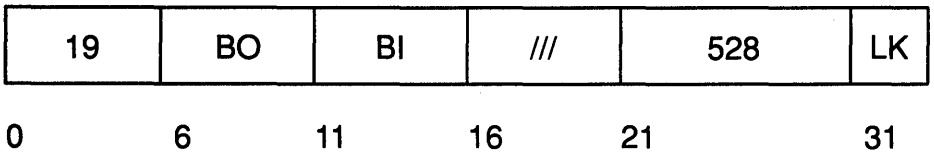
Conditionally branches to the address contained within the Count Register.

Syntax

bcc *BO,BI*

bccl *BO,BI*

Extended mnemonics are also provided.



Description

The **bcc** instruction conditionally branches to an instruction specified by the branch target address contained within the Count Register. The branch target address is the concatenation of Count Register bits 0–29 and b'00'.

The Branch Option (BO) field has one of the following specifications:

BO	Description
0000x	Decrement the Count Register, then branch if the decremented CTR is not 0 and condition FALSE.
0001x	Decrement the Count Register, then branch if the decremented CTR is 0 and condition FALSE.
001xx	Branch if condition FALSE.
0100x	Decrement the Count Register, then branch if the decremented CTR is not 0 and condition TRUE.
0101x	Decrement the Count Register, then branch if the decremented CTR is 0 and condition TRUE.
011xx	Branch if condition TRUE.
1x00x	Decrement the Count Register, then branch if the decremented CTR is not 0.
1x01x	Decrement the Count Register, then branch if the decremented CTR is 0.
1x1xx	Branch always.

The **bcc** instruction has two syntax forms. Each syntax form has a different effect on the Link bit and Link Register.

Syntax form	Absolute Address bit (AA)	Fixed Point Exception Register	Link bit (LK)	Condition Register Field 0
bcc	None	None	0	None
bccl	None	None	1	None

The two syntax forms of the **bcc** instruction never affect the Fixed Point Exception Register or Condition Register Field 0. If the Link bit is 1, then the effective address of the instruction following the branch instruction is placed into the Link Register.

Parameters

BO Specifies Branch Option field.

BI Specifies bit in Condition Register for condition comparison.

BIF Specifies the Condition Register field that specifies the Condition Register bit (LT, GT, EQ, SO) to be used for condition comparison.

Extended Mnemonics

Two extended mnemonic branch commands are based on the **bcc** command. Use the **l** form to place the instruction that follows the Branch Instruction in the Link Register, and use the 12 Branch Codes to replace **XX** in each Extended Mnemonic command.

Syntax	Parameters	Description
bctr	None	Branch to Count Register
bXXc, bXXcl	[*BIF*]	Branch Count Register on XX Condition

Examples

1. To branch from a specific address, dependant on a bit in the Condition Register, to the address contained in the Count Register:

```
bcc 0x4,0
cror 15,15,15
# Branch occurs if LT bit in the Condition Register is 0.
# The branch will be to the address contained in
# the Count Register.
bccl 0xC,1
return: cror 15,15,15
# Branch occurs if GT bit in the Condition Register is 1.
# The branch will be to the address contained in
# the Count Register.
# The Link register now contains the address of return.
```

Implementation Specifics

This instruction is part of Application Development Toolkit in AIX Base.

Related Information

See the following article in *POWERstation and POWERserver Hardware Technical Reference — General Information*: Count Register.

Understanding Branch Instructions on page 1–3.

bcr (Branch Conditional Register) Instruction

Purpose

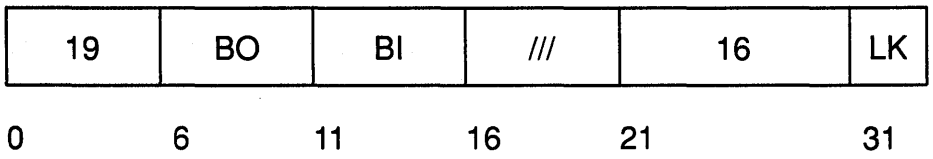
Conditionally branches to an address contained in the Link Register.

Syntax

bcr *BO,BI*

bcrl *BO,BI*

Extended mnemonics are also provided.



Description

The **bcr** instruction branches to an instruction specified by the branch target address. The branch target address is the concatenation of bits 0–29 of the Link Register and b'00'.

The Branch Option (BO) field has one of the following specifications:

BO	Description
0000x	Decrement the Count Register, then branch if the decremented CTR is not 0 and condition FALSE.
0001x	Decrement the Count Register, then branch if the decremented CTR is 0 and condition FALSE.
001xx	Branch if condition FALSE.
0100x	Decrement the Count Register, then branch if the decremented CTR is not 0 and condition TRUE.
0101x	Decrement the Count Register, then branch if the decremented CTR is 0 and condition TRUE.
011xx	Branch if condition TRUE.
1x00x	Decrement the Count Register, then branch if the decremented CTR is not 0.
1x01x	Decrement the Count Register, then branch if the decremented CTR is 0.
1x1xx	Branch always.

The **bcr** instruction has two syntax forms. Each syntax form has a different effect on the Link bit and Link Register.

Syntax form	Absolute Address bit (AA)	Fixed Point Exception Register	Link bit (LK)	Condition Register Field 0
bcr	None	None	0	None
bcrl	None	None	1	None

The two syntax forms of the **bcr** instruction never affect the Fixed Point Exception Register or Condition Register Field 0. If the Link bit (LK) is 1, then the effective address of the instruction following the branch instruction is placed into the Link Register.

Parameters

BO Specifies Branch Option field.

BI Specifies bit in Condition Register for condition comparison.

BIF Specifies the Condition Register field that specifies the Condition Register bit (LT, GT, EQ, SO) to be used for condition comparison.

Extended Mnemonics

One extended mnemonic branch command is based on the **bcr** command. Use the **l** form to place the instruction that follows the Branch Instruction in the Link Register, and use the 12 Branch Codes to replace **XX** in the command.

Syntax	Parameters	Description
bXXr, bXXrl	[*BIF*]	Branch Register on Condition

Examples

1. To branch to the calculated branch target address dependant on bit 0 of the Condition Register:

```
bcr 0x0,0
# The Count Register is decremented.
# A branch occurs if the LT bit is set to zero in the
# Condition Register and if the Count Register
# does not equal zero.
# If the conditions are met, the instruction branches to
# the concatination of bits 0–29 of the Link Register and b'00'.
```

Implementation Specifics

This instruction is part of Application Development Toolkit in AIX Base.

Related Information

See the following article in *POWERstation and POWERserver Hardware Technical Reference — General Information*: Count Register.

Understanding Branch Instructions on page 1–3.

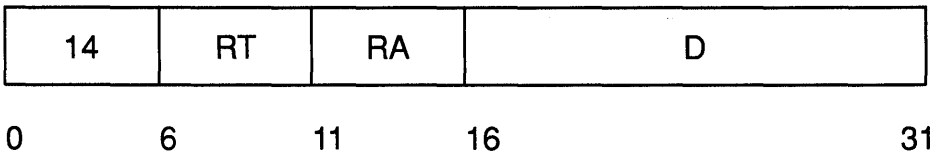
cal (Compute Address Lower) Instruction

Purpose

Calculates an address from an offset and a base address and places the result in a general purpose register.

Syntax

cal *RT,D(RA)*



Description

The **cal** instruction places the sum of the contents of General Purpose Register *RA* and the 16–bit two's complement integer *D*, sign extended to 32 bits, into the target General Purpose Register *RT*. If General Purpose Register *RA* is GPR 0, then *D* is stored into the target General Purpose Register *RT*.

The **cal** instruction has one syntax form and does not affect Condition Register Field 0 or the Fixed Point Exception Register.

Parameters

RT	Specifies target general purpose register where result of operation is stored.
RA	Specifies source general purpose register for operation.
D	Specifies 16–bit two's complement integer sign extended to 32 bits.

Examples

1. To calculate an address or contents with an offset of 0xFFFF 8FF0 from the contents of GPR 5 and store the result in GPR 4:

```
# Assume GPR 5 contains 0x0000 0900.
cal 4,0xFFFF8FF0(5)
# GPR 4 now contains 0xFFFF 98F0.
```

Implementation Specifics

This instruction is part of Application Development Toolkit in AIX Base.

Related Information

Understanding Fixed Point Address Computation Instructions on page 1–8.

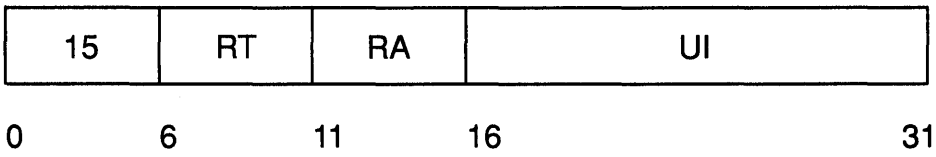
cau (Compute Address Upper) Instruction

Purpose

Calculates an address from a concatenated offset and a base address and loads the result in a general purpose register.

Syntax

cau *RT,RA,UI*



Description

The **cau** instruction places the sum of the contents of General Purpose Register *RA* and the concatenation of a 16-bit unsigned integer *UI* and x'0000' into the target General Purpose Register *RT*. If General Purpose Register *RA* is GPR 0, then the sum of the concatenation of *UI* and x'0000' and zero is stored into the target General Purpose Register *RT*.

The **cau** instruction has one syntax form and does not affect Condition Register Field 0 or the Fixed Point Exception Register.

Parameters

RT	Specifies target general purpose register where result of operation is stored.
RA	Specifies first source general purpose register for operation.
UI	Specifies concatenation of 16–bit unsigned integer for operation.

Examples

1. To add an offset of 0x0011 0000 to the address or contents contained in GPR 6 and load the result into GPR 7:

```
# Assume GPR 6 contains 0x0000 4000.
cau 7,6,0x0011
# GPR 7 now contains 0x0011 4000.
```

Implementation Specifics

This instruction is part of Application Development Toolkit in AIX Base.

Related Information

Understanding Fixed Point Address Computation Instructions on page 1–8.

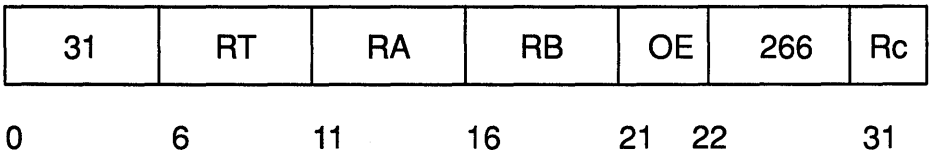
cax (Compute Address) Instruction

Purpose

Calculates an address by adding the contents of two general purpose registers and places the result in a general purpose register.

Syntax

cax *RT,RA,RB*
cax. *RT,RA,RB*
caxo *RT,RA,RB*
caxo. *RT,RA,RB*



Description

The **cax** instruction places the sum of the contents of General Purpose Register *RA* and General Purpose Register *RB* into the target General Purpose Register *RT*.

The **cax** instruction has four syntax forms. Each syntax form has a different effect on Condition Register Field 0 and the Fixed Point Exception Register.

Syntax form	Overflow Exception (OE)	Fixed Point Exception Register	Record bit (Rc)	Condition Register Field 0
cax	0	None	0	None
cax.	0	None	1	LT,GT,EQ,SO
caxo	1	SO,OV	0	None
caxo.	1	SO,OV	1	LT,GT,EQ,SO

The four syntax forms of the **cax** instruction never affect the Carry bit (CA) in the Fixed Point Exception Register. If the syntax form sets the Overflow Exception (OE) bit to 1, the instruction affects the Summary Overflow (SO) and Overflow (OV) bits in the Fixed Point Exception Register. If the syntax form sets the Record (Rc) bit to 1, the instruction affects the Less Than (LT) zero, Greater Than (GT) zero, Equal To (EQ) zero, and Summary Overflow (SO) bits in Condition Register Field 0.

Parameters

RT Specifies target general purpose register where result of operation is stored.

RA Specifies source general purpose register for operation.

RB Specifies source general purpose register for operation.

Examples

1. To add the address or contents in GPR 6 to the address or contents in GPR 3 and store the result in GPR 4:

```
# Assume GPR 6 contains 0x0004 0000.
# Assume GPR 3 contains 0x0000 4000.
cax 4,6,3
# GPR 4 now contains 0x0004 4000.
```

2. To add the address or contents in GPR 6 to the address or contents in GPR 3, store the result in GPR 4, and set Condition Register Field 0 to reflect the result of the operation:

```
# Assume GPR 6 contains 0x8000 7000.
# Assume GPR 3 contains 0x7000 8000.
 cax. 4,6,3
# GPR 4 now contains 0xF000 F000.
```

3. To add the address or contents in GPR 6 to the address or contents in GPR 3, store the result in GPR 4, and set the Summary Overflow, Overflow, and Carry bits in the Fixed Point Exception Register to reflect the result of the operation:

```
# Assume GPR 6 contains 0xEFFF FFFF.
# Assume GPR 3 contains 0x8000 0000.
caxo 4,6,3
# GPR 4 now contains 0x6FFF FFFF.
```

4. To add the address or contents in GPR 6 to the address or contents in GPR 3, store the result in GPR 4, and set the Summary Overflow, Overflow, and Carry bits in the Fixed Point Exception Register and Condition Register Field 0 to reflect the result of the operation:

```
# Assume GPR 6 contains 0xEFFF FFFF.
# Assume GPR 3 contains 0xEFFF FFFF.
caxo. 4,6,3
# GPR 4 now contains 0xDFFF FFFE.
```

Implementation Specifics

This instruction is part of Application Development Toolkit in AIX Base.

Related Information

Understanding Fixed Point Address Computation Instructions on page 1–8.

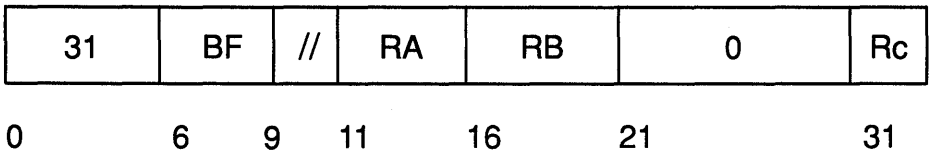
cmp (Compare) Instruction

Purpose

Compares the contents of two general purpose registers algebraically.

Syntax

cmp *BF,RA,RB*



Description

The **cmp** instruction compares the contents of General Purpose Register *RA* with the contents of General Purpose Register *RB* as signed integers and sets one of the Condition Register Field *BF* bits.

BF can be Condition Register Field 0–7; programmers can specify which Condition Register Field will indicate the result of the operation.

The Condition Register Field *BF* bits are interpreted as follows:

Bit	Name	Description
0	LT	(RA) < SI
1	GT	(RA) > SI
2	EQ	(RA) = SI
3	SO	SO,OV

The **cmp** instruction has one syntax form and does not affect the Fixed Point Exception Register. Condition Register Field 0 is unaffected unless it is specified as *BF* by the programmer.

Parameters

BF Specifies Condition Register Field 0–7 that indicates result of compare.

RA Specifies source general purpose register for operation.

RB Specifies source general purpose register for operation.

Examples

1. To compare the the contents of GPR 4 and GPR 6 as signed integers and set Condition Register Field 0 to reflect the result of the operation:

```
# Assume GPR 4 contains 0xFFFF FFE7.
# Assume GPR 5 contains 0x0000 0011.
# Assume 0 is Condition Register Field 0.
cmp 0,4,6
# The LT bit of Condition Register Field 0 is set.
```

Implementation Specifics

This instruction is part of Application Development Toolkit in AIX Base.

Related Information

The **cmpi** (Compare Immediate) instruction, **cmpl** (Compare Logical) instruction, **cmpli** (Compare Logical Immediate) instruction.

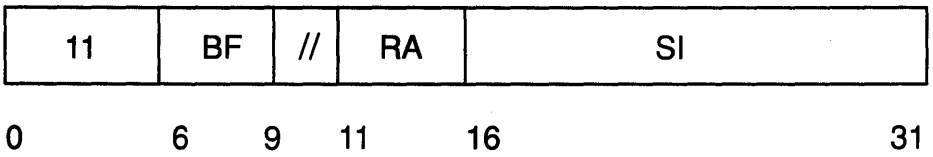
cmpi (Compare Immediate) Instruction

Purpose

Compares the contents of a general purpose register and a given value algebraically.

Syntax

cmpi *BF,RA,SI*



Description

The **cmpi** instruction compares the contents of General Purpose Register *RA* and a sixteen bit signed integer *SI* as signed integers and sets one of the Condition Register Field *BF* bits.

BF can be Condition Register Field 0–7; programmers can specify which Condition Register Field will indicate the result of the operation.

The Condition Register Field *BF* bits are interpreted as follows.

Bit	Name	Description
0	LT	(RA) < SI
1	GT	(RA) > SI
2	EQ	(RA) = SI
3	SO	SO,OV

The **cmp** instruction has one syntax form and does not affect the Fixed Point Exception Register. Condition Register Field 0 is unaffected unless it is specified as *BF* by the programmer.

Parameters

BF	Specifies Condition Register Field 0–7 that indicates result of compare.
RA	Specifies first source general purpose register for operation.
SI	Specifies 16–bit signed integer for operation.

Examples

1. To compare the the contents of GPR 4 and the signed integer 0x11 and set Condition Register Field 0 to reflect the result of the operation:

```
# Assume GPR 4 contains 0xFFFF FFE7.
cmpi 0,4,0x11
# The LT bit of Condition Register Field 0 is set.
```

Implementation Specifics

This instruction is part of Application Development Toolkit in AIX Base.

Related Information

The **cmp** (Compare) instruction, **cmpl** (Compare Logical) instruction, **cmpli** (Compare Logical Immediate) instruction.

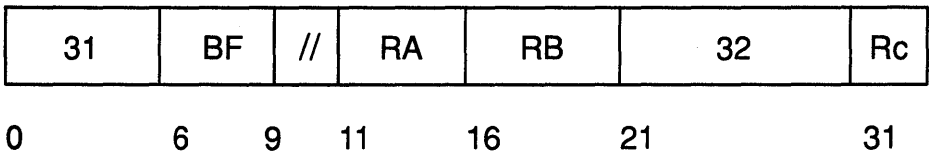
cmpl (Compare Logical) Instruction

Purpose

Compares the contents of two general purpose registers logically.

Syntax

cmpl *BF,RA,RB*



Description

The **cmpl** instruction compares the contents of General Purpose Register *RA* with the contents of General Purpose Register *RB* as unsigned integers and sets one of the Condition Register Field *BF* bits.

BF can be Condition Register Field 0–7; programmers can specify which Condition Register Field will indicate the result of the operation.

The Condition Register Field *BF* bits are interpreted as follows:

Bit	Name	Description
0	LT	(RA) < SI
1	GT	(RA) > SI
2	EQ	(RA) = SI
3	SO	SO,OV

The **cmpl** instruction has one syntax form and does not affect the Fixed Point Exception Register. Condition Register Field 0 is unaffected unless it is specified as *BF* by the programmer.

Parameters

BF Specifies Condition Register Field 0–7 indicates result of compare.

RA Specifies source general purpose register for operation.

RB Specifies source general purpose register for operation.

Examples

1. To compare the the contents of GPR 4 and GPR 5 as unsigned integers and set Condition Register Field 0 to reflect the result of the operation:

```
# Assume GPR 4 contains 0xFFFF 0000.
# Assume GPR 5 contains 0x7FFF 0000.
# Assume 0 is Condition Register Field 0.
cmpl 0,4,5
# The GT bit of Condition Register Field 0 is set.
```

Implementation Specifics

This instruction is part of Application Development Toolkit in AIX Base.

Related Information

The **cmp** (Compare) instruction, **cmpi** (Compare Immediate) instruction, **cmpli** (Compare Logical Immediate) instruction.

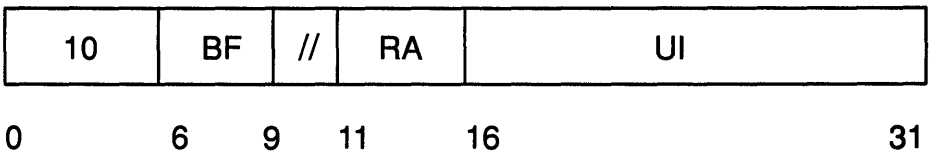
cmpli (Compare Logical Immediate) Instruction

Purpose

Compares the contents of a general purpose register and a given value logically.

Syntax

cmpli *BF,RA,UI*



Description

The **cmpli** instruction compares the contents of General Purpose Register *RA* with the concatenation of x'0000' and a 16–bit unsigned integer *UI* as unsigned integers and sets one of the Condition Register Field *BF* bits.

BF can be Condition Register Field 0–7; programmers can specify which Condition Register Field will indicate the result of the operation.

The Condition Register Field *BF* bits are interpreted as follows:

Bit	Name	Description
0	LT	(RA) < SI
1	GT	(RA) > SI
2	EQ	(RA) = SI
3	SO	SO,OV

The **cmpli** instruction has one syntax form and does not affect the Fixed Point Exception Register. Condition Register Field 0 is unaffected unless it is specified as *BF* by the programmer.

Parameters

BF	Specifies Condition Register Field 0–7 that indicates result of compare.
RA	Specifies source general purpose register for operation.
UI	Specifies 16–bit unsigned integer for operation.

Examples

1. To compare the contents of GPR 4 and the unsigned integer 0xff and set Condition Register Field 0 to reflect the result of the operation:

```
# Assume GPR 4 contains 0x0000 00ff.
cmpli 0,4,0xff
# The EQ bit of Condition Register Field 0 is set.
```

Implementation Specifics

This instruction is part of Application Development Toolkit in AIX Base.

Related Information

The **cmp** (Compare) instruction, **cmpi** (Compare Immediate) instruction, **cmpl** (Compare Logical) instruction.

cntlz (Count Leading Zeros) Instruction

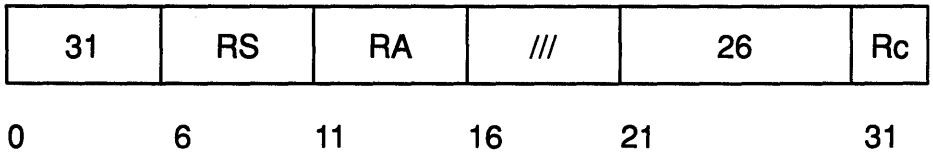
Purpose

Places the number of leading zeros from a source general purpose register in a general purpose register.

Syntax

cntlz *RA,RS*

cntlz. *RA,RS*



Description

The **cntlz** instruction counts the number (between 0 and 32 inclusive) of consecutive zero bits starting at bit zero of General Purpose Register *RS* and stores the result in the target General Purpose Register *RA*.

Syntax form		Fixed Point Exception Register	Record bit (Rc)	Condition Register Field 0
cntlz	None	None	0	None
cntlz.	None	None	1	LT,GT,EQ,SO

The two syntax forms of the **cntlz** instruction never affect the Fixed Point Exception Register. If the syntax form sets the Record (Rc) bit to 1, the instruction affects the Less Than (LT) zero, Greater Than (GT) zero, Equal To (EQ) zero, and Summary Overflow (SO) bits in Condition Register Field 0.

Parameters

RA Specifies target general purpose register where result of operation is stored.

RS Specifies source general purpose register for operation.

Examples

1. To count the number of leading zeros in the value contained in GPR 3 and place the result back in GPR 3:

```
# Assume GPR 3 contains 0x0061 9920.
cntlz 3,3
# GPR 3 now holds 0x0000 0009.
```

Implementation Specifics

This instruction is part of Application Development Toolkit in AIX Base.

Related Information

Understanding Fixed Point Logical Instructions on page 1–9.

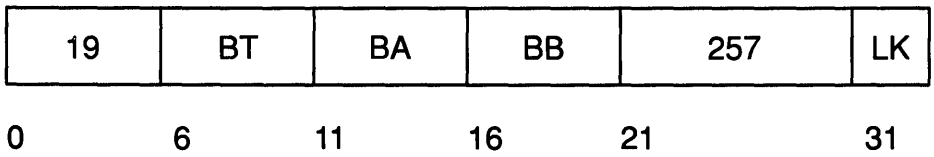
crand (Condition Register AND) Instruction

Purpose

Places the result of ANDing two Condition Register bits in a Condition Register bit.

Syntax

crand *BT,BA,BB*



Description

The **crand** instruction ANDs the Condition Register bit specified by *BA* and the Condition Register bit specified by *BB* and places the result in the target Condition Register bit specified by *BT*.

The **crand** instruction has one syntax form and does not affect the Fixed Point Exception Register.

Parameters

BT	Specifies target Condition Register bit where result of operation is stored.
BA	Specifies source Condition Register bit for operation.
BB	Specifies source Condition Register bit for operation.

Examples

1. To AND Condition Register bits 0 and 5 and store the result in Condition Register bit 31:

```
# Assume Condition Register bit 0 is 1.
# Assume Condition Register bit 5 is 0.
crand 31,0,5
# Condition Register bit 31 is now 0.
```

Implementation Specifics

This instruction is part of Application Development Toolkit in AIX Base.

Related Information

See the following article in *POWERstation and POWERserver Hardware Technical Reference — General Information*: Condition Register.

Understanding Condition Register Instructions on page 1–5.

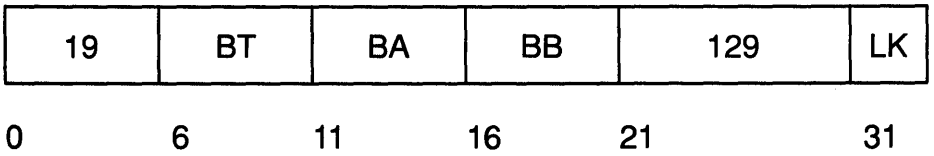
crandc (Condition Register AND with Complement) Instruction

Purpose

Places the result of ANDing one Condition Register bit and the complement of a Condition Register bit in a Condition Register bit.

Syntax

crandc *BT,BA,BB*



Description

The **crandc** instruction ANDs the Condition Register bit specified in *BA* and the complement of the Condition Register bit specified by *BB* and places the result in the target Condition Register bit specified by *BT*.

The **crand** instruction has one syntax form and does not affect the Fixed Point Exception Register.

Parameters

BT	Specifies target Condition Register bit where result of operation is stored.
BA	Specifies source Condition Register bit for operation.
BB	Specifies source Condition Register bit for operation.

Examples

1. To AND Condition Register bit 0 and the complement of Condition Register bit 5 and put the result in bit 31:

```
# Assume Condition Register bit 0 is 1.
# Assume Condition Register bit 5 is 0.
crandc 31,0,5
# Condition Register bit 31 is now 1.
```

Implementation Specifics

This instruction is part of Application Development Toolkit in AIX Base.

Related Information

See the following article in *POWERstation and POWERserver Hardware Technical Reference — General Information*: Condition Register.

Understanding Condition Register Instructions on page 1–5.

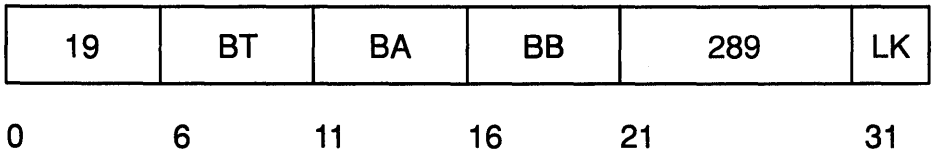
creqv (Condition Register Equivalent) Instruction

Purpose

Places the complemented result of XORing two Condition Register bits in a Condition Register bit.

Syntax

creqv *BT,BA,BB*



Description

The **creqv** instruction XORs the Condition Register bit specified in *BA* and the Condition Register bit specified by *BB* and places the complemented result in the target Condition Register bit specified by *BT*.

The **creqv** instruction has one syntax form and does not affect the Fixed Point Exception Register.

Parameters

BT	Specifies target Condition Register bit where result of operation is stored.
BA	Specifies source Condition Register bit for operation.
BB	Specifies source Condition Register bit for operation.

Examples

1. To place the complemented result of XORing Condition Register bits 8 and 4 into Condition Register bit 4:

```
# Assume Condition Register bit 8 is 1.
# Assume Condition Register bit 4 is 0.
creqv 4,8,4
# Condition Register bit 4 is now 0.
```

Implementation Specifics

This instruction is part of Application Development Toolkit in AIX Base.

Related Information

See the following article in *POWERstation and POWERserver Hardware Technical Reference — General Information*: Condition Register.

Understanding Condition Register Instructions on page 1–5.

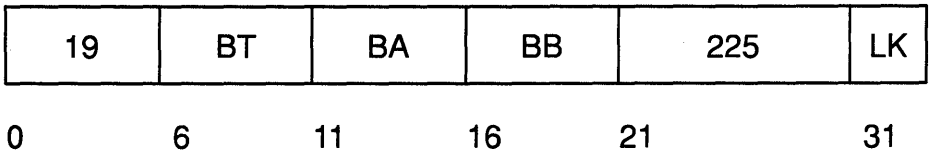
crnand (Condition Register NAND) Instruction

Purpose

Places the complemented result of ANDing two Condition Register bits in a Condition Register bit.

Syntax

crnand *BT,BA,BB*



Description

The **crnand** instruction ANDs the Condition Register bit specified by *BA* and the Condition Register bit specified by *BB* and places the complemented result in the target Condition Register bit specified by *BT*.

The **crnand** instruction has one syntax form and does not affect the Fixed Point Exception Register.

Parameters

BT	Specifies target Condition Register bit where result of operation is stored.
BA	Specifies source Condition Register bit for operation.
BB	Specifies source Condition Register bit for operation.

Examples

1. To ANDing Condition Register bits 8 and 4 and place the complemented result into Condition Register bit 4:

```
# Assume Condition Register bit 8 is 1.
# Assume Condition Register bit 4 is 0.
crnand 4,8,4
# Condition Register bit 4 is now 1.
```

Implementation Specifics

This instruction is part of Application Development Toolkit in AIX Base.

Related Information

See the following article in *POWERstation and POWERserver Hardware Technical Reference — General Information*: Condition Register.

Understanding Condition Register Instructions on page 1–5.

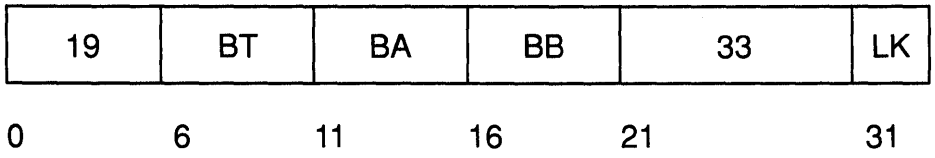
crnor (Condition Register NOR) Instruction

Purpose

Places the complemented result of ORing two Condition Register bits in a Condition Register bit.

Syntax

crnor *BT,BA,BB*



Description

The **crnor** instruction ORs the Condition Register bit specified in *BA* and the Condition Register bit specified by *BB* and places the complemented result in the target Condition Register bit specified by *BT*.

The **crnor** instruction has one syntax form and does not affect the Fixed Point Exception Register.

Parameters

BT	Specifies target Condition Register bit where result of operation is stored.
BA	Specifies source Condition Register bit for operation.
BB	Specifies source Condition Register bit for operation.

Examples

1. To OR Condition Register bits 8 and 4 and store the complemented result into Condition Register bit 4:

```
# Assume Condition Register bit 8 is 1.
# Assume Condition Register bit 4 is 0.
crnor 4,8,4
# Condition Register bit 4 is now 0.
```

Implementation Specifics

This instruction is part of Application Development Toolkit in AIX Base.

Related Information

See the following article in *POWERstation and POWERserver Hardware Technical Reference — General Information*: Condition Register.

Understanding Condition Register Instructions on page 1–5.

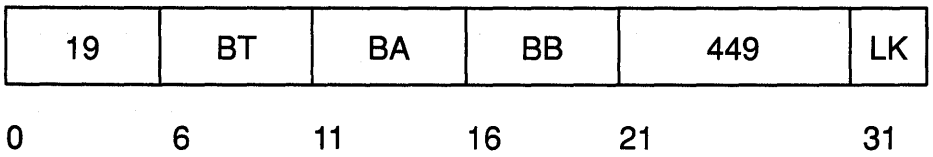
cror (Condition Register OR) Instruction

Purpose

Places the result of ORing two Condition Register bits in a Condition Register bit.

Syntax

cror *BT,BA,BB*



Description

The **cror** instruction ORs the Condition Register bit specified by *BA* and the Condition Register bit specified by *BB* and places the result in the target Condition Register bit specified by *BT*.

The **cror** instruction has one syntax form and does not affect the Fixed Point Exception Register.

Parameters

BT	Specifies target Condition Register bit where result of operation is stored.
BA	Specifies source Condition Register bit for operation.
BB	Specifies source Condition Register bit for operation.

Examples

1. To place the result of ORing Condition Register bits 8 and 4 into Condition Register bit 4:

```
# Assume Condition Register bit 8 is 1.
# Assume Condition Register bit 4 is 0.
cror 4,8,4
# Condition Register bit 4 is now 1.
```

Implementation Specifics

This instruction is part of Application Development Toolkit in AIX Base.

Related Information

See the following article in *POWERstation and POWERserver Hardware Technical Reference — General Information*: Condition Register.

Understanding Condition Register Instructions on page 1–5.

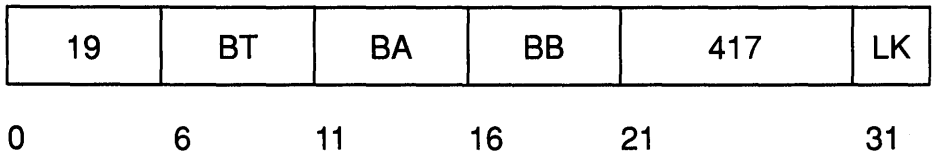
crorc (Condition Register OR with Complement) Instruction

Purpose

Places the result of ORing a Condition Register bit and the complement of a Condition Register bit in a Condition Register bit.

Syntax

crorc *BT,BA,BB*



Description

The **crorc** instruction ORs the Condition Register bit specified by *BA* and the complement of the Condition Register bit specified by *BB* and places the result in the target Condition Register bit specified by *BT*.

The **crorc** instruction has one syntax form and does not affect the Fixed Point Exception Register.

Parameters

BT	Specifies target Condition Register bit where result of operation is stored.
BA	Specifies source Condition Register bit for operation.
BB	Specifies source Condition Register bit for operation.

Examples

1. To place the result of ORing Condition Register bit 8 and the complement of Condition Register bit 4 into Condition Register bit 4:

```
# Assume Condition Register bit 8 is 1.
# Assume Condition Register bit 4 is 0.
crorc 4,8,4
# Condition Register bit 4 is now 1.
```

Implementation Specifics

This instruction is part of Application Development Toolkit in AIX Base.

Related Information

See the following article in *POWERstation and POWERserver Hardware Technical Reference — General Information*: Condition Register.

Understanding Condition Register Instructions on page 1–5.

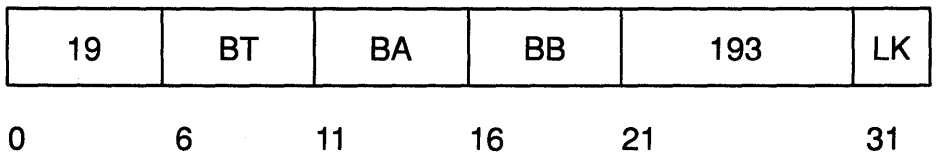
crxor (Condition Register XOR) Instruction

Purpose

Places the result of XORing two Condition Register bits in a Condition Register bit.

Syntax

crxor *BT,BA,BB*



Description

The **crxor** instruction XORs the Condition Register bit specified by *BA* and the Condition Register bit specified by *BB* and places the result in the target Condition Register bit specified by *BT*.

The **crxor** instruction has one syntax form and does not affect the Fixed Point Exception Register.

Parameters

BT	Specifies target Condition Register bit where result of operation is stored.
BA	Specifies source Condition Register bit for operation.
BB	Specifies source Condition Register bit for operation.

Examples

1. To place the result of XORing Condition Register bits 8 and 4 into Condition Register bit 4:

```
# Assume Condition Register bit 8 is 1.
# Assume Condition Register bit 4 is 1.
crxor 4,8,4
# Condition Register bit 4 is now 0.
```

Implementation Specifics

This instruction is part of Application Development Toolkit in AIX Base.

Related Information

See the following article in *POWERstation and POWERserver Hardware Technical Reference — General Information*: Condition Register.

Understanding Condition Register Instructions on page 1–5.

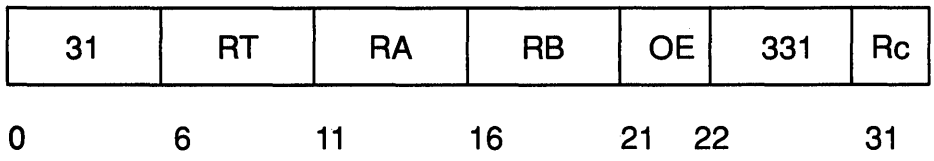
div (Divide) Instruction

Purpose

Divides the contents of a general purpose register concatenated with the MQ Register by the contents of a general purpose register and stores the result in a general purpose register.

Syntax

div *RT,RA,RB*
div. *RT,RA,RB*
divo *RT,RA,RB*
divo. *RT,RA,RB*



Description

The **div** instruction concatenates the contents of General Purpose Register *RA* and the contents of Multiply Quotient (MQ) Register, divides the result by the contents of General Purpose Register *RB*, and stores the result in the target General Purpose Register *RT*. The remainder has the same sign as the dividend, except a zero quotient or a zero remainder is always positive. The results obey the equation

dividend = (divisor x quotient) + remainder

where a dividend is the original (RA) || (MQ), divisor is the original (*RB*), quotient is the final (*RT*), and remainder is the final (MQ).

For the case of –2**31 ÷ –1, the MQ Register is set to zero and –2**31 is placed in General Purpose Register *RT*. For all other overflows, the contents of MQ, the target General Purpose Register *RT*, and the Condition Register Field 0 (if the Record Bit (Rc) is 1) are undefined.

The **div** instruction has four syntax forms. Each syntax form has a different effect on Condition Register Field 0 and the Fixed Point Exception Register.

Syntax form		Fixed Point Exception Register	Record bit (Rc)	Condition Register Field 0
div	0	None	0	None
div.	0	None	1	LT,GT,EQ,SO
divo	1	SO,OV	0	None
divo.	1	SO,OV	1	LT,GT,EQ,SO

The four syntax forms of the **div** instruction never affect the Carry bit (CA) in the Fixed Point Exception Register. If the syntax form sets the Overflow Exception (OE) bit to 1, the instruction affects the Summary Overflow (SO) and Overflow (OV) bits in the Fixed Point Exception Register. If the syntax form sets the Record (Rc) bit to 1, the instruction affects the Less Than (LT) zero, Greater Than (GT) zero, Equal To (EQ) zero, and Summary Overflow (SO) bits in Condition Register Field 0.

Parameters

RT	Specifies target general purpose register where result of operation is stored.
RA	Specifies source general purpose register for operation.
RB	Specifies source general purpose register for operation.

Examples

1. To divide the contents of GPR 4, concatenated with the MQ register, by the contents of GPR 6 and store the result in GPR 4:

```
# Assume the MQ Register contains 0x0000 0001.
# Assume GPR 4 contains 0x0000 0000.
# Assume GPR 6 contains 0x0000 0002.
div 4,4,6
# GPR 4 now contains 0x0000 0000.
# The MQ register now contains 0x0000 0001.
```

2. To divide the contents of GPR 4, concatenated with the MQ register, by the contents of GPR 6, store the result in GPR 4 and set Condition Register Field 0 to reflect the result of the operation:

```
# Assume the MQ Register contains 0x0000 0002.
# Assume GPR 4 contains 0x0000 0002.
# Assume GPR 6 contains 0x0000 0002.
div. 4,4,6
# GPR 4 now contains 0x0000 0001.
# MQ contains 0x0000 0000.
```

3. To divide the contents of GPR 4, concatenated with the MQ register, by the contents of GPR 6, place the result in GPR 4, and set the Summary Overflow and Overflow bits in the Fixed Point Exception Register to reflect the result of the operation:

```
# Assume GPR 4 contains 0x0000 0001.
# Assume GPR 6 contains 0x0000 0000.
# Assume the MQ Register contains 0x0000 0000.
divo 4,4,6
# GPR 4 now contains an undefined quantity.
# The MQ register is undefined.
```

4. To divide the contents of GPR 4, concatenated with the MQ register, by the contents of GPR 6, place the result in GPR 4, and set the Summary Overflow and Overflow bits in the Fixed Point Exception Register and Condition Register Field 0 to reflect the result of the operation:

```
# Assume GPR 4 contains 0x-1.
# Assume GPR 6 contains 0x2.
# Assume the MQ Register contains 0xFFFFFFFF.
divo. 4,4,6
# GPR 4 now contains 0x0000 0000.
# The MQ register contains 0x-1.
```

Implementation Specifics

This instruction is part of Application Development Toolkit in AIX Base.

Related Information

Understanding Fixed Point Arithmetic Instructions on page 1–8.

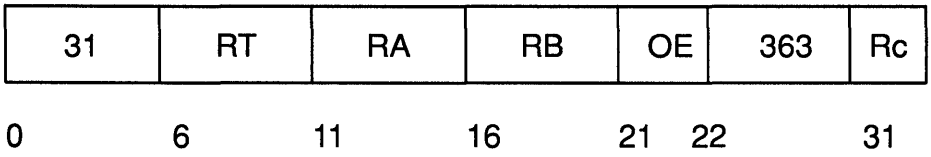
divs (Divide Short) Instruction

Purpose

Divides the contents of a general purpose register by the contents of a general purpose register and stores the result in a general purpose register.

Syntax

divs *RT,RA,RB*
divs. *RT,RA,RB*
divso *RT,RA,RB*
divso. *RT,RA,RB*



Description

The **divs** instruction divides the contents of General Purpose Register *RA* by the contents of General Purpose Register *RB* and stores the result in the target General Purpose Register *RT*. The remainder has the same sign as the dividend, except a zero quotient or a zero remainder is always positive. The results obey the equation

dividend = (divisor x quotient) + remainder

where a dividend is the original (*RA*), divisor is the original (*RB*), quotient is the final (*RT*), and remainder is the final (MQ).

For the case of –2**31 ÷ –1, the MQ Register is set to zero and –2**31 is placed in General Purpose Register *RT*. For all other overflows, the contents of MQ, the target General Purpose Register *RT* and the Condition Register Field 0 (if the Record Bit (Rc) is 1) are undefined.

The **divs** instruction has four syntax forms. Each syntax form has a different effect on Condition Register Field 0 and the Fixed Point Exception Register.

Syntax form	Overflow Exception (OE)	Fixed Point Exception Register	Record bit (Rc)	Condition Register Field 0
divs	0	None	0	None
divs.	0	None	1	LT,GT,EQ,SO
divso	1	SO,OV	0	None
divso.	1	SO,OV	1	LT,GT,EQ,SO

The four syntax forms of the **divs** instruction never affect the Carry bit (CA) in the Fixed Point Exception Register. If the syntax form sets the Overflow Exception (OE) bit to 1, the instruction affects the Summary Overflow (SO) and Overflow (OV) bits in the Fixed Point Exception Register. If the syntax form sets the Record (Rc) bit to 1, the instruction affects the Less Than (LT) zero, Greater Than (GT) zero, Equal To (EQ) zero, and Summary Overflow (SO) bits in Condition Register Field 0.

Parameters

RT	Specifies target general purpose register where result of operation is stored.
RA	Specifies source general purpose register for operation.
RB	Specifies source general purpose register for operation.

Examples

1. To divide the contents of GPR 4 by the contents of GPR 6 and store the result in GPR 4:

```
# Assume GPR 4 contains 0x0000 0001.
# Assume GPR 6 contains 0x0000 0002.
divs 4,4,6
# GPR 4 now contains 0x0.
# The MQ register now contains 0x1.
```

2. To divide the contents of GPR 4 by the contents of GPR 6, store the result in GPR 4 and set Condition Register Field 0 to reflect the result of the operation:

```
# Assume GPR 4 contains 0x0000 0002.
# Assume GPR 6 contains 0x0000 0002.
divs. 4,4,6
# GPR 4 now contains 0x0000 0001.
# The MQ register now contains 0x0000 0000.
```

3. To divide the contents of GPR 4 by the contents of GPR 6, store the result in GPR 4, and set the Summary Overflow and Overflow bits in the Fixed Point Exception Register to reflect the result of the operation:

```
# Assume GPR 4 contains 0x0000 0001.
# Assume GPR 6 contains 0x0000 0000.
divso 4,4,6
# GPR 4 now contains an undefined quantity.
```

4. To divide the contents of GPR 4 by the contents of GPR 6, store the result in GPR 4, and set the Summary Overflow and Overflow bits in the Fixed Point Exception Register and Condition Register Field 0 to reflect the result of the operation:

```
# Assume GPR 4 contains 0x-1.
# Assume GPR 6 contains 0x0000 00002.
# Assume the MQ Register contains 0x0000 0000.
divso. 4,4,6
# GPR 4 now contains 0x0000 0000.
# The MQ register contains 0x-1.
```

Implementation Specifics

This instruction is part of Application Development Toolkit in AIX Base.

Related Information

Understanding Fixed Point Arithmetic Instructions on page 1–8.

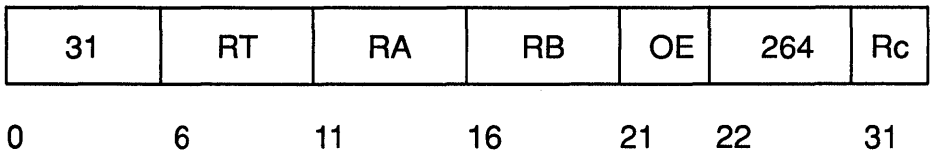
doz (Difference or zero) Instruction

Purpose

Computes the difference between the contents of two general purpose registers and stores the result or the value zero in a general purpose register.

Syntax

doz *RT,RA,RB*
doz. *RT,RA,RB*
dozo *RT,RA,RB*
dozo. *RT,RA,RB*



Description

The **doz** instruction adds the complement of the contents of General Purpose Register *RA*, 1, and the contents of General Purpose Register *RB* and stores the result in the target General Purpose Register *RT*.

If the value in General Purpose Register *RA* is algebraically greater than the value in General Purpose Register *RB*, then General Purpose Register *RT* is set to zero.

The **doz** instruction has four syntax forms. Each syntax form has a different effect on Condition Register Field 0 and the Fixed Point Exception Register.

Syntax form	Overflow Exception (OE)	Fixed Point Exception Register	Record bit (Rc)	Condition Register Field 0
doz	0	None	0	None
doz.	0	None	1	LT,GT,EQ,SO
dozo	1	SO,OV	0	None
dozo.	1	SO,OV	1	LT,GT,EQ,SO

The four syntax forms of the **doz** instruction never affect the Carry bit (CA) in the Fixed Point Exception Register. If the syntax form sets the Overflow Exception (OE) bit to 1, the instruction affects the Summary Overflow (SO) and Overflow (OV) bits in the Fixed Point Exception Register; the Overflow (OV) bit can only be set on positive overflows. If the syntax form sets the Record (Rc) bit to 1, the instruction effects the Less Than (LT) zero, Greater Than (GT) zero, Equal To (EQ) zero, and Summary Overflow (SO) bits in Condition Register Field 0.

Parameters

RT	Specifies target general purpose register where result of operation is stored.
RA	Specifies source general purpose register for operation.
RB	Specifies source general purpose register for operation.

Examples

1. To determine the difference between the contents of GPR 4 and GPR 6 and store the result in GPR 4:

```
# Assume GPR 4 holds 0x0000 0001.
# Assume GPR 6 holds 0x0000 0002.
doz 4,4,6
# GPR 4 now holds 0x0000 0001.
```

2. To determine the difference between the contents of GPR 4 and GPR 6, store the result in GPR 4, and set Condition Register Field 0 to reflect the result of the operation:

```
# Assume GPR 4 holds 0x0000 0001.
# Assume GPR 6 holds 0x0000 0000.
doz. 4,4,6
# GPR 4 now holds 0x0000 0000.
```

3. To determine the difference between the contents of GPR 4 and GPR 6, store the result in GPR 4, and set set the Summary Overflow and Overflow bits in the Fixed Point Exception Register to reflect the result of the operation:

```
# Assume GPR 4 holds 0x0000 0002.
# Assume GPR 6 holds 0x0000 0008.
dozo 4,4,6
# GPR 4 now holds 0x0000 0006.
```

4. To determine the difference between the contents of GPR 4 and GPR 6, store the result in GPR 4, and set the the Summary Overflow and Overflow bits in the Fixed Point Exception Register and Condition Register Field 0 to reflect the result of the operation:

```
# Assume GPR 4 holds 0xEFFF FFFF.
# Assume GPR 6 holds 0x0000 0000.
dozo. 4,4,6
# GPR 4 now holds 0x1000 0001.
```

Implementation Specifics

This instruction is part of Application Development Toolkit in AIX Base.

Related Information

Understanding Fixed Point Arithmetic Instructions on page 1–8.

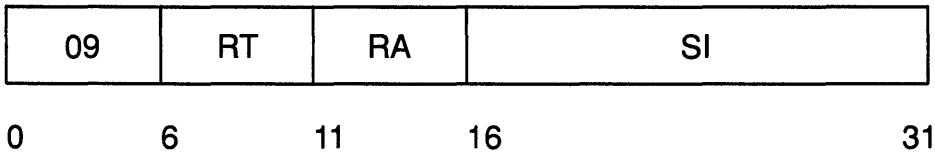
dozi (Difference or Zero Immediate) Instruction

Purpose

Computes the difference between the contents of a general purpose register and a signed 16-bit integer and stores the result or the value zero in a general purpose register.

Syntax

dozi *RT,RA,SI*



Description

The **dozi** instruction adds the complement of the contents of General Purpose Register *RA*, the 16-bit signed integer *SI*, and 1 and stores the result in the target General Purpose Register *RT*.

If the value in General Purpose Register *RA* is algebraically greater than the 16–bit signed value in the *SI* field, then General Purpose Register *RT* is set to zero.

The **dozi** instruction has one syntax form and does not effect Condition Register Field 0 or the Fixed Point Exception Register.

Parameters

RT	Specifies target general purpose register where result of operation is stored.
RA	Specifies source general purpose register for operation.
SI	Specifies signed 16–bit integer for operation.

Examples

1. To determine the difference between GPR 4 and 0x0 and store the result in GPR 4:

```
# Assume GPR 4 holds 0x0000 0001.
dozi 4,4,0x0
# GPR 4 now holds 0x0000 0000.
```

Implementation Specifics

This instruction is part of Application Development Toolkit in AIX Base.

Related Information

Understanding Fixed Point Arithmetic Instructions on page 1–8.

eqv (Equivalent) Instruction

Purpose

XORs the contents of two general purpose registers and places the complemented result in a general purpose register.

Syntax

eqv *RA,RS,RB*

eqv. *RA,RS,RB*

Description

The **eqv** instruction XORs the contents of General Purpose Register *RS* with the contents of General Purpose Register *RB* and stores the complemented result in the target General Purpose Register *RA*.

The **eqv** instruction has two syntax forms. Each syntax form has a different effect on Condition Register Field 0.

Syntax form	Overflow Exception (OE)	Fixed Point Exception Register	Record bit (Rc)	Condition Register Field 0
eqv	None	None	0	None
eqv.	None	None	1	LT,GT,EQ,SO

The two syntax forms of the **eqv** instruction never affect the Fixed Point Exception Register. If the syntax form sets the Record (Rc) bit to 1, the instruction affects the Less Than (LT) zero, Greater Than (GT) zero, Equal To (EQ) zero, and Summary Overflow (SO) bits in Condition Register Field 0.

Parameters

RT Specifies target general purpose register where result of operation is stored.

RA Specifies source general purpose register for operation.

RB Specifies source general purpose register for operation.

Examples

1. To XOR the contents of GPR 4 and GPR 6 and store the complemented result in GPR 4:

```
# Assume GPR 4 holds 0xFFF2 5730.
# Assume GPR 6 holds 0x7B41 92C0.
eqv 4,4,6
# GPR 4 now holds 0x7B4C 3A0F.
```

2. To XOR the contents of GPR 4 and GPR 6, store the complemented result in GPR 4, and set Condition Register Field 0 to reflect the result of the operation:

```
# Assume GPR 4 holds 0x0000 00FD.
# Assume GPR 6  holds 0x7B41 92C0.
eqv. 4,4,6
# GPR 4 now holds 0x84BE 6DC2.
```

Implementation Specifics

This instruction is part of Application Development Toolkit in AIX Base.

Related Information

Understanding Fixed Point Logical Instructions on page 1–9.

exts (Extend Sign) Instruction

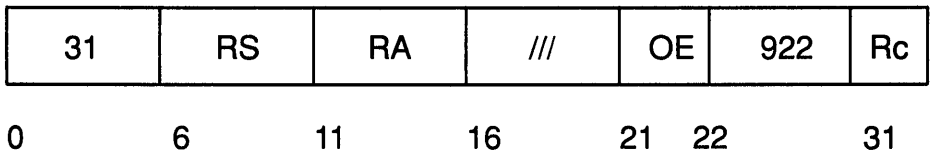
Purpose

Extends the lower 16-bit contents of a general purpose register.

Syntax

exts *RA,RS*

exts. *RA,RS*



Description

The **exts** instruction places bits 16–31 of General Purpose Register *RS* in bits 16–31 of General Purpose Register *RA* and copies bit 16 of register *RS* in bits 0–15 of register *RA*.

The **exts** instruction has two syntax forms. Each syntax form has a different effect on Condition Register Field 0.

Syntax form	Overflow Exception (OE)	Fixed Point Exception Register	Record bit (Rc)	Condition Register Field 0
exts	None	None	0	None
exts.	None	None	1	LT,GT,EQ,SO

The two syntax forms of the **exts** instruction never affect the Fixed Point Exception Register. If the syntax form sets the Record (Rc) bit to 1, the instruction affects the Less Than (LT) zero, Greater Than (GT) zero, Equal To (EQ) zero, and Summary Overflow (SO) bits in Condition Register Field 0.

Parameters

RA Specifies general purpose register receives extended integer.

RS Specifies source general purpose register for operation.

Examples

1. To place bits 16–31 of GPR 6 into bits 16–31 of GPR 4 and copy bit 16 of GPR 6 into bits 0–15 of GPR 4:

```
# Assume GPR 6 holds 0x0000 FFFF.
exts 4,6
# GPR 6 now holds 0xFFFF FFFF.
```

2. To place bits 16–31 of GPR 6 into bits 16–31 of GPR 4, copy bit 16 of GPR 6 into bits 0–15 of GPR 4, and set Condition Register Field 0 to reflect the result of the operation:

```
# Assume GPR 4 holds 0x0000 2FFF.
exts. 6,4
# GPR 6 now holds 0x0000 2FFF.
```

Implementation Specifics

This instruction is part of Application Development Toolkit in AIX Base.

Related Information

Understanding Fixed Point Logical Instructions on page 1–9.

fa (Floating Add) Instruction

Purpose

Adds two 64-bit double precision floating point operands and places the result in a Floating Point register.

Syntax

fa *FRT,FRA,FRB*
fa. *FRT,FRA,FRB*

Description

The **fa** instruction adds the 64-bit double precision floating point operand in Floating Point Register *FRA* to the 64-bit double precision floating point operand in Floating Point Register *FRB*. The result is rounded under control of the Floating Point Rounding Control Field *RN* of the Floating Point Status and Control Register and is placed in Floating Point register *FRT*.

Addition of two floating point numbers is based on exponent comparison and addition of the two significands. The exponents of the two operands are compared, and the significand accompanying the smaller exponent is shifted right, with its exponent increased by one for each bit shifted, until the two exponents are equal. The two significands are then added algebraically to form the intermediate sum. All 53 bits in the significand as well as all three guard bits (G,R and X) enter into the computation.

Tininess is checked before rounding. The unrounded result is then rounded using the mode specified by the RM field of the Floating Point Status and Control Register. The rounded result is then checked for overflow and inexact exceptions.

- If the sum of two operands with opposite signs is exactly zero, then the sign of that sum is positive in all rounding modes except Round Toward –Infinity, in which mode that sign is negative. The sum of operands with the same sign retains the sign of the operands, even if the operands are zeros.

The Floating Point Result Field of the Floating Point Status and Control Register is set to the class and sign of the result except for Invalid Operation Exceptions when the Floating Point Invalid Operation Exception Enable (VE) bit of the Floating Point Status and Control Register is set to 1.

The **fa** instruction has two syntax forms. Each syntax form has a different effect on Condition Register Field 1.

Syntax form	Floating Point Status and Control Register	Record bit (Rc)	Condition Register Field 1
fa	C,FL,FG,FE,FU,FR,FI,OX,UX, XX,VXSNAN,VXISI	0	None
fa.	C,FL,FG,FE,FU,FR,FI,OX,UX, XX,VXSNAN,VXISI	1	FX,FEX,VX,OX

The two syntax forms of the **fa** instruction always affect the Floating Point Status and Control Register. If the syntax form sets the Record (Rc) bit to 1, the instruction affects the Floating Point Exception Summary (FX), Floating Point Enabled Exception Summary (FEX), Floating Point Invalid Operation Exception Summary (VX), and Floating Point Overflow Exception (OX) bits in Condition Register Field 1.

Parameters

FRT Specifies target Floating Point register for operation.

FRA Specifies source Floating Point register for operation.

FRB Specifies source Floating Point register for operation.

Examples

1. To add the contents of FPR 4 and FPR 5, place the result in FPR 6, and set the Floating Point Status and Control Register to reflect the result of the operation:

```
# Assume FPR 4 contains 0xC053 4000 0000 0000.
# Assume FPR 5 contains 0x400C 0000 0000 0000.
# Assume RM = 0.
fa 6,4,5
# FPR 6 now contains 0xC052 6000 0000 0000.
```

2. To add the contents of FPR 4 and FPR 25, place the result in FPR 6, and set Condition Register Field 1 and the Floating Point Status and Control Register to reflect the result of the operation:

```
# Assume FPR 4 contains 0xC053 4000 0000 0000.
# Assume FPR 25 contains 0xFFFF FFFF FFFF FFFF.
# Assume RM = 0.
fa. 6,4,25
# GPR 6 now contains 0xFFFF FFFF FFFF FFFF.
```

Implementation Specifics

This instruction is part of Application Development Toolkit in AIX Base.

Related Information

Understanding Floating Point Arithmetic Instructions on page 1–14.

Understanding The Floating Point Status and Control Register on page 1–12.

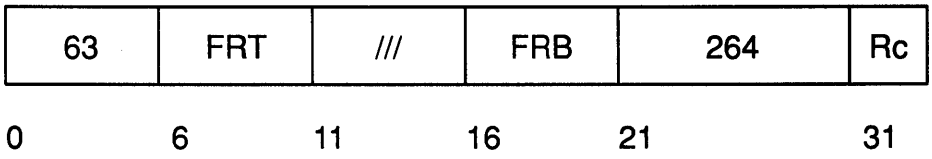
fabs (Floating Absolute Value) Instruction

Purpose

Stores the absolute value of the contents of a Floating Point register in a Floating Point register.

Syntax

fabs *FRT,FRB*
fabs. *FRT,FRB*



Description

The **fabs** instruction sets bit 0 of Floating Point Register *FRB* to zero and places the result into Floating Point Register *FRT*.

The **fabs** instruction has two syntax forms. Each syntax form has a different effect on Condition Register Field 1.

Syntax form	Floating Point Status and Control Register	Record bit (Rc)	Condition Register Field 1
fabs	None	0	None
fabs.	None	1	FX,FEX,VX,OX

The two syntax forms of the **fabs** instruction never affectthe Floating Point Status and Control Register. If the syntax form sets the Record (Rc) bit to 1, the instruction affects the Floating Point Exception Summary (FX), Floating Point Enabled Exception Summary (FEX), Floating Point Invalid Operation Exception Summary (VX), and Floating Point Overflow Exception (OX) bits in Condition Register Field 1.

Parameters

FRT Specifies target Floating Point register for operation.

FRB Specifies source Floating Point register for operation.

Examples

1. To set bit 0 of FPR 4 to zero and place the result in FPR 6:

```
# Assume FPR 4 holds 0xC053 4000 0000 0000.
fabs 6,4
# GPR 6 now holds 0x4053 4000 0000 0000.
```

2. To set bit 0 of FPR 25 to zero, place the result in FPR 6, and set Condition Register Field 1 to reflect the result of the operation:

```
# Assume FPR 25 holds 0xFFFF FFFF FFFF FFFF.
fabs. 6,25
# GPR 6 now holds 0x7FFF FFFF FFFF FFFF.
```

Implementation Specifics

This instruction is part of Application Development Toolkit in AIX Base.

Related Information

Understanding Floating Point Move Instructions on page 1–13.

Understanding The Floating Point Status and Control Register on page 1–12.

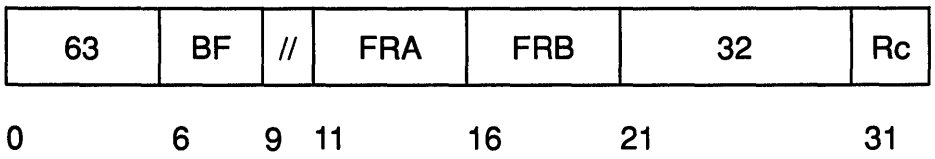
fcmpo (Floating Compare Ordered) Instruction

Purpose

Compares the contents of two Floating Point registers.

Syntax

fcmpo *BF,FRA,FRB*



Description

The **fcmpo** instruction compares the 64-bit double precision floating point operand in Floating Point Register *FRA* to the 64-bit double precision floating point operand in Floating Point Register *FRB*. The Floating Point Condition Code Field (FPCC) of the Floating Point Status and Control Register (FPSCR) is set to reflect the value of the operand *FRA* with respect to operand *FRB*. The value *BF* determines which field in the Condition Register receives the four FPCC bits.

- If one of the operands is either a Quiet Nan or a Signaling NaN, the Floating Point Condition Code is set to reflect unordered (FU).
- If one of the operands is a Signaling NaN, then the Floating Point Invalid Operation Exception bit VXSNAN of the Floating Point Status and Control Register is set. Also:
  - If Invalid Operation is disabled (i.e., the Floating Point Invalid Operation Exception Enable bit of the Floating Point Status and Control Register is 0), then the Floating Point Invalid Operation Exception bit VXVC is set (signaling an an Invalid compare).
  - If one of the operands is a Quiet NaN, then the Floating Point Invalid Operation Exception bit VXVC is set.

The **fcmpo** instruction has one syntax form and always affects the FT, FG, FE and FU VXSNAN, and VXVC bits in the Floating Point Status and Control Register (FPSCR).

Parameters

BF	Specifies field in the Condition Register that receives the four FPCC bits.
FRA	Specifies source Floating Point register.
FRB	Specifies source Floating Point register.

Examples

1. To compare the contents of FPR 4 and FPR 6 and and set Condition Register Field 1 and the Floating Point Status and Control Register to reflect the result of the operation:

```
# Assume CR = 0 and FPSCR = 0.
# Assume FPR 5 contains 0xC053 4000 0000 0000.
# Assume FPR 4 contains 0x400C 0000 0000 0000.
fcmpo 6,4,5
# CR now contains 0x0000 0040.
# FPSCR now contains 0x0000 4000.
```

Implementation Specifics

This instruction is part of Application Development Toolkit in AIX Base.

Related Information

Understanding Floating Point Compare Instructions on page 1–14.

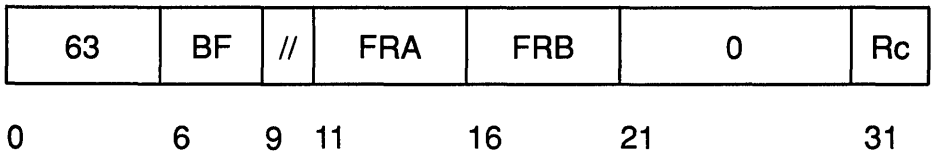
fcmpu (Floating Compare Unordered) Instruction

Purpose

Compares the contents of two Floating Point registers.

Syntax

fcmpu *BF,FRA,FRB*



Description

The **fcmpu** instruction compares the 64-bit double precision floating point operand in Floating Point Register *FRA* to the 64-bit double precision floating point operand in Floating Point Register *FRB*. The Floating Point Condition Code Field (FPCC) of the Floating Point Status and Control Register (FPSCR) is set to reflect the value of the operand *FRA* with respect to operand *FRB*. The value *BF* determines which field in the Condition Register receives the four FPCC bits.

- If one of the operands is either a Quiet NaN or a Signaling NaN, the Floating Point Condition Code is set to reflect unordered (FU).
- If one of the operands is a Signaling NaN, then the Floating Point Invalid Operation Exception bit VXSNAN of the Floating Point Status and Control Register is set.

The **fcmpu** instruction has one syntax form and always affects the FT, FG, FE and FU and VXSNAN bits in the Floating Point Status and Control Register (FPSCR).

Parameters

BF	Specifies field in the Condition Register that receives the four FPCC bits.
FRA	Specifies source Floating Point register.
FRB	Specifies source Floating Point register.

Examples

1. To compare the contents of FPR 5 and FPR 4:

```
# Assume FPR 5 holds 0xC053 4000 0000 0000.
# Assume FPR 4 holds 0x400C 0000 0000 0000.
# Assume CR = 0 and FPSCR = 0.
fcmpu 6,4,5
# CR now contains 0x0000 0040.
# FPSCR now contains 0x0000 4000.
```

Implementation Specifics

This instruction is part of Application Development Toolkit in AIX Base.

Related Information

Understanding Floating Point Compare Instructions on page 1–14.

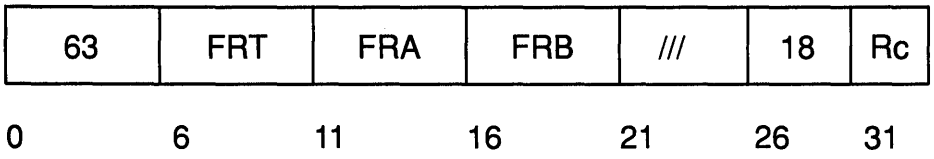
fd (Floating Divide) Instruction

Purpose

Divides one 64-bit double precision floating point operand by another.

Syntax

fd *FRT,FRA,FRB*
fd. *FRT,FRA,FRB*



Description

The **fd** instruction divides the 64-bit double precision floating point operand in Floating Point Register *FRA* by the 64-bit double precision floating point operand in Floating Point Register *FRB*. No remainder is preserved. The result is rounded under control of the Floating Point Rounding Control Field *RN* of the Floating Point Status and Control Register (FPSCR) and is placed in the target Floating Point register *FRT*.

The floating point division operation is based on exponent subtraction and division of the two significands.

- If an operand is a denormalized number, then it is prenormalized before the operation is begun.

The Floating Point Result Flags field of the Floating Point Status and Control Register is set to the class and sign of the result except for Invalid Operation Exceptions when the Floating Point Invalid Operation Exception Enable bit is 1.

The **fd** instruction has two syntax forms. Each syntax form has a different effect on Condition Register Field 1.

Syntax form	Floating Point Status and Control Register	Record bit (Rc)	Condition Register Field 1
fd	C,FL,FG,FE,FU,FR,FI,OX,UX, ZX,XX,VXSNAN,VXIDI,VXZDZ	0	None
fd.	C,FL,FG,FE,FU,FR,FI,OX,UX, ZX,XX,VXSNAN,VXIDI,VXZDZ	1	FX,FEX,VX,OX

The two syntax forms of the **fd** instruction always affect the Floating Point Status and Control Register. If the syntax form sets the Record (Rc) bit to 1, the instruction affects the Floating Point Exception (FX), Floating Point Enabled Exception (FEX), Floating Point Invalid Operation Exception (VX), and Floating Point Overflow Exception (OX) bits in Condition Register Field 1.

Parameters

FRT Specifies target Floating Point register for operation.

FRA Specifies source Floating Point register containing the dividend.

FRB Specifies source Floating Point register containing the divisor.

Examples

1. To divide the contents of FPR 4 by the contents of FPR 5, place the result in FPR 6, and set the Floating Point Status and Control Register to reflect the result of the operation:

```
# Assume FPR 4 contains 0xC053 4000 0000 0000.
# Assume FPR 5 contains 0x400C 0000 0000 0000.
# Assume RM = 0 and FPSCR = 0.
fd 6,4,5
# FPR 6 now contains 0xC036 0000 0000 0000.
# FPSCR now contains 0x0000 8000.
```

2. To divide the contents of FPR 4 by the contents of FPR 5, place the result in FPR 6, and set Condition Register Field 1 and the Floating Point Status and Control Register to reflect the result of the operation:

```
# Assume FPR 4 contains 0xC053 4000 0000 0000.
# Assume FPR 5 contains 0x400C 0000 0000 0000.
# Assume RM = 0 and FPSCR = 0.
fd. 6,4,5
# FPR 6 now contains 0xC036 0000 0000 0000.
# FPSCR now contains 0x0000 8000.
# CR contains 0x0000 0000.
```

Implementation Specifics

This instruction is part of Application Development Toolkit in AIX Base.

Related Information

Understanding Floating Point Arithmetic Instructions on page 1–14.

Understanding The Floating Point Status and Control Register on page 1–12.

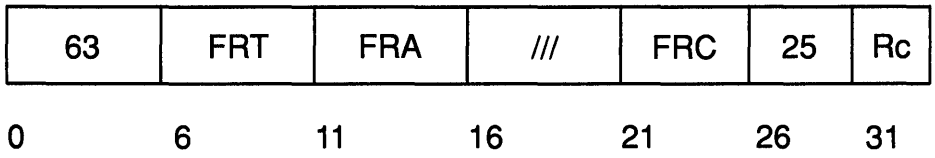
fm (Floating Multiply) Instruction

Purpose

Multiplies two 64-bit double precision floating point operands.

Syntax

fm *FRT,FRA,FRC*
fm. *FRT,FRA,FRC*



Description

The **fm** instruction multiplies the 64-bit double precision floating point operand in Floating Point Register *FRA* by the 64-bit double precision floating point operand in Floating Point Register *FRC*. The result is rounded under control of the Floating Point Rounding Control Field *RN* of the Floating Point Status and Control Register and is placed in the target Floating Point Register *FRT*.

Multiplication of two floating point numbers is based on exponent addition and multiplication of the two significands.

- If an operand is a denormalized number then it is prenormalized before the operation is begun.

The Floating Point Result Flags field of the Floating Point Status and Control Register is set to the class and sign of the result except for Invalid Operation Exceptions when the Floating Point Invalid Operation Exception Enable bit is 1.

The **fm** instruction has two syntax forms. Each syntax form has a different effect on Condition Register Field 1.

Syntax form	Floating Point Status and Control Register	Record bit (Rc)	Condition Register Field 1
fm	C,FL,FG,FE,FU,FR,FI,OX,UX, XX,VXSNAN,VXIMZ	0	None
fm.	C,FL,FG,FE,FU,FR,FI,OX,UX, XX,VXSNAN,VXIMZ	1	FX,FEX,VX,OX

The two syntax forms of the **fm** instruction always affect the Floating Point Status and Control Register. If the syntax form sets the Record (Rc) bit to 1, the instruction affects the Floating Point Exception (FX), Floating Point Enabled Exception (FEX), Floating Point Invalid Operation Exception (VX), and Floating Point Overflow Exception (OX) bits in Condition Register Field 1.

Parameters

FRT Specifies target Floating Point register for operation.

FRA Specifies source Floating Point register for operation.

FRC Specifies source Floating Point register for operation.

Examples

1. To multiply the contents of FPR 4 and FPR 5, place the result in FPR 6, and set the Floating Point Status and Control Register to reflect the result of the operation:

```
# Assume FPR 4 contains 0xC053 4000 0000 0000.
# Assume FPR 5 contains 0x400C 0000 0000 0000.
# Assume RM = 0 and FPSCR = 0.
fm 6,4,5
# FPR 6 now contains 0xC070 D800 0000 0000.
# FPSCR now contains 0x0000 8000.
```

2. To multiply the contents of FPR 4 and FPR 25, place the result in FPR 6, and set Condition Register Field 1 and the Floating Point Status and Control Register to reflect the result of the operation:

```
# Assume FPR  4 contains 0xC053 4000 0000 0000.
# Assume FPR 25 contains 0xFFFF FFFF FFFF FFFF.
# Assume RM = 0, FPSCR = 0, and CR = 0.
fm. 6,4,25
# FPR 6 now contains 0xFFFF FFFF FFFF FFFF.
# FPSCR now contains 0x0001 1000.
# CR now contains 0x0000 0000.
```

Implementation Specifics

This instruction is part of Application Development Toolkit in AIX Base.

Related Information

Understanding Floating Point Arithmetic Instructions on page 1–14.

Understanding The Floating Point Status and Control Register on page 1–12.

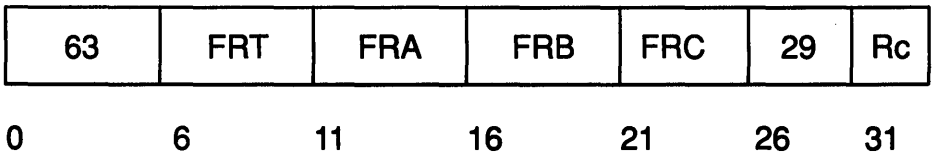
fma (Floating Multiply Add) Instruction

Purpose

Adds one 64-bit double precision floating point operand to the result of multiplying two 64-bit double precision floating point operands without an intermediate rounding operation.

Syntax

fma *FRT,FRA,FRC,FRB*
fma. *FRT,FRA,FRC,FRB*



Description

The **fma** instruction multiplies the 64-bit double precision floating point operand in Floating Point Register *FRA* by the 64-bit double precision floating point operand in Floating Point Register *FRC* and adds result of this operation to the 64-bit double precision floating point operand in Floating Point Register *FRB*.

The result is rounded under control of the Floating Point Rounding Control Field *RN* of the Floating Point Status and Control Register and is placed in the target Floating Point register *FRT*.

- If an operand is a denormalized number, then it is prenormalized before the operation is begun.

The Floating Point Result Flags field of the Floating Point Status and Control Register is set to the class and sign of the result except for Invalid Operation Exceptions when the Floating Point Invalid Operation Exception Enable bit is 1.

The **fma** instruction has two syntax forms. Each syntax form has a different effect on Condition Register Field 1.

Syntax form	Floating Point Status and Control Register	Record bit (Rc)	Condition Register Field 1
fma	C,FL,FG,FE,FU,FR,FI,OX,UX, XX,VXSNAN,VXISI,VXIMZ	0	None
fma.	C,FL,FG,FE,FU,FR,FI,OX,UX, XX,VXSNAN,VXISI,VXIMZ	1	FX,FEX,VX,OX

The two syntax forms of the **fma** instruction always affect the Floating Point Status and Control Register. If the syntax form sets the Record (Rc) bit to 1, the instruction affects the Floating Point Exception (FX), Floating Point Enabled Exception (FEX), Floating Point Invalid Operation Exception (VX), and Floating Point Overflow Exception (OX) bits in Condition Register Field 1.

Parameters

FRT Specifies target Floating Point register for operation.

FRA Specifies source Floating Point register containing a multiplier.

FRB Specifies source Floating Point register containing the addend.

FRC Specifies source Floating Point register containing a multiplier.

Examples

1. To multiply the contents of FPR 4 and FPR 5, add the contents of FPR 7, place the result in FPR 6, and set the Floating Point Status and Control Register to reflect the result of the operation:

```
# Assume FPR 4 contains 0xC053 4000 0000 0000.
# Assume FPR 5 contains 0x400C 0000 0000 0000.
# Assume FPR 7 contains 0x3DE2 6AB4 B33C 110A.
# Assume RM = 0 and FPSCR = 0.
fma 6,4,5,7
# FPR 6 now contains 0xC070 D7FF FFFF F6CB.
# FPSCR now contains 0x8206 8000.
```

2. To multiply the contents of FPR 4 and FPR 5, add the contents of FPR 7, place the result in FPR 6, and set the Floating Point Status and Control Register and Condition Register Field 1 to reflect the result of the operation:

```
# Assume FPR 4 contains 0xC053 4000 0000 0000.
# Assume FPR 5 contains 0x400C 0000 0000 0000.
# Assume FPR 7 contains 0x3DE2 6AB4 B33C 110A.
# Assume RM = 0, FPSCR = 0, and CR = 0.
fma. 6,4,5,7
# FPR 6 now contains 0xC070 D7FF FFFF F6CB.
# FPSCR now contains 0x8206 8000.
# CR now contains 0x0800 0000.
```

Implementation Specifics

This instruction is part of Application Development Toolkit in AIX Base.

Related Information

Understanding Floating Point Accumulate Instructions on page 1–14.

Understanding The Floating Point Status and Control Register on page 1–12.

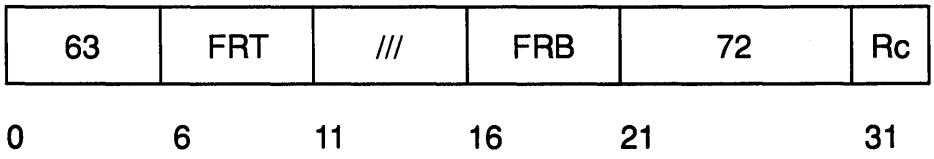
fmr (Floating Move Register) Instruction

Purpose

Copies the contents of one Floating Point register into another Floating Point register.

Syntax

fmr *FRT,FRB*
fmr. *FRT,FRB*



Description

The **fmr** instruction places the contents of Floating Point Register *FRB* into the target Floating Point Register *FRT*.

The **fmr** instruction has two syntax forms. Each syntax form has a different effect on Condition Register Field 1.

Syntax form	Floating Point Status and Control Register	Record bit (Rc)	Condition Register Field 1
fmr	None	0	None
fmr.	None	1	FX,FEX,VX,OX

The two syntax forms of the **fmr** instruction never affect the Floating Point Status and Control Register. If the syntax form sets the Record (Rc) bit to 1, the instruction affects the Floating Point Exception (FX), Floating Point Enabled Exception (FEX), Floating Point Invalid Operation Exception (VX), and Floating Point Overflow Exception (OX) bits in Condition Register Field 1.

Parameters

FRT Specifies target Floating Point register for operation.

FRB Specifies source Floating Point register for operation.

Examples

1. To copy the contents of FPR 4 into FPR 6 and set the Floating Point Status and Control Register to reflect the result of the operation:

```
# Assume FPR 4 contains 0xC053 4000 0000 0000.
# Assume FPSCR = 0.
fmr 6,4
# FPR 6 now contains 0xC053 4000 0000 0000.
# FPSCR now contains 0x0000 0000.
```

2. To copy the contents of FPR 25 into FPR 6 and set the Floating Point Status and Control Register and Condition Register Field 1 to reflect the result of the operation:

```
# Assume FPR 25 contains 0xFFFF FFFF FFFF FFFF.
# Assume FPSCR = 0 and CR = 0.
fmr. 6,25
# FPR 6 now contains 0xFFFF FFFF FFFF FFFF.
# FPSCR now contains 0x0000 0000.
# CR now contains 0x0000 0000.
```

Implementation Specifics

This instruction is part of Application Development Toolkit in AIX Base.

Related Information

Understanding Floating Point Move Instructions on page 1–13.

Understanding The Floating Point Status and Control Register on page 1–12.

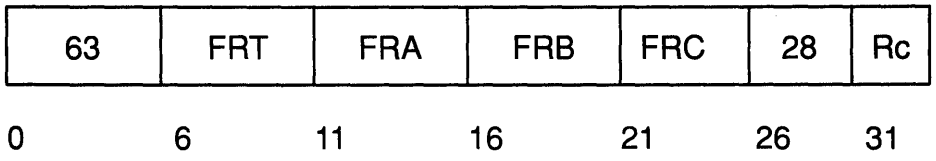
fms (Floating Multiply Subtract) Instruction

Purpose

Subtracts one 64-bit double precision floating point operand from the result of multiplying two 64-bit double precision floating point operands without an intermediate rounding operation.

Syntax

fms *FRT,FRA,FRC,FRB*
fms. *FRT,FRA,FRC,FRB*



Description

The **fms** instruction multiplies the 64-bit double precision floating point operand in Floating Point Register *FRA* by the 64-bit double precision floating point operand in Floating Point Register *FRC* and subtracts the 64-bit double precision floating point operand in Floating Point Register *FRB* from the result of the multiplication.

The result is rounded under control of the Floating Point Rounding Control Field *RN* of the Floating Point Status and Control Register and is placed in the target Floating Point Register *FRT*.

- If an operand is a denormalized number, then it is prenormalized before the operation is begun.

The Floating Point Result Flags field of the Floating Point Status and Control Register is set to the class and sign of the result except for Invalid Operation Exceptions when the Floating Point Invalid Operation Exception Enable bit is 1.

The **fms** instruction has two syntax forms. Each syntax form has a different effect on Condition Register Field 1.

Syntax form	Floating Point Status and Control Register	Record bit (Rc)	Condition Register Field 1
fms	C,FL,FG,FE,FU,FR,FI,OX,UX, XX,VXSNAN,VXSI,VXIMZ	0	None
fms.	C,FL,FG,FE,FU,FR,FI,OX,UX, XX,VXSNAN,VXSI,VXIMZ	1	FX,FEX,VX,OX

The two syntax forms of the **fms** instruction always affect the Floating Point Status and Control Register. If the syntax form sets the Record (Rc) bit to 1, the instruction affects the Floating Point Exception (FX), Floating Point Enabled Exception (FEX), Floating Point Invalid Operation Exception (VX), and Floating Point Overflow Exception (OX) bits in Condition Register Field 1.

Parameters

FRT	Specifies target Floating Point register for operation.
FRA	Specifies source Floating Point register containing a multiplier.
FRB	Specifies source Floating Point register containing the quantity to be subtracted.
FRC	Specifies source Floating Point register containing a multiplier.

Examples

1. To multiply the contents of FPR 4 and FPR 5, subtract the the contents of FPR 7 from the product of the multiplication, place the result in FPR 6, and set the Floating Point Status and Control Register to reflect the result of the operation:

```
# Assume FPR 4 contains 0xC053 4000 0000 0000.
# Assume FPR 5 contains 0x400C 0000 0000 0000.
# Assume FPR 7 contains 0x3DE2 6AB4 B33c 110A.
# Assume RM = 0 and FPSCR = 0.
fms 6,4,5,7
# FPR 6 now contains 0xC070 D800 0000 0935.
# FPSCR now contains 0x8202 8000.
```

2. To multiply the contents of FPR 4 and FPR 5, subtract the the contents of FPR 7 from the product of the multiplication, place the result in FPR 6, and set the Floating Point Status and Control Register and Condition Register Field 1 to reflect the result of the operation:

```
# Assume FPR 4 contains 0xC053 4000 0000 0000.
# Assume FPR 5 contains 0x400C 0000 0000 0000.
# Assume FPR 7 contains 0x3DE2 6AB4 B33c 110A.
# Assume RM = 0, FPSCR = 0, and CR = 0.
fms. 6,4,5,7
# FPR 6 now contains 0xC070 D800 0000 0935.
# FPSCR now contains 0x8202 8000.
# CR now contains 0x0800 0000.
```

Implementation Specifics

This instruction is part of Application Development Toolkit in AIX Base.

Related Information

Understanding Floating Point Accumulate Instructions on page 1–14.

Understanding The Floating Point Status and Control Register on page 1–12.

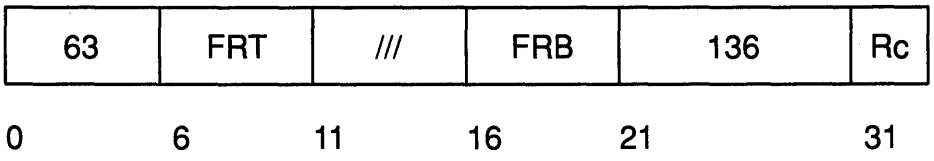
fnabs (Floating Negative Absolute Value) Instruction

Purpose

Negates the absolute contents of a Floating Point register and places the result in a Floating Point register.

Syntax

fnabs *FRT,FRB*
fnabs. *FRT,FRB*



Description

The **fnabs** instruction places the negative absolute of the contents of Floating Point Register *FRB* with bit 0 set to 1 into the target Floating Point Register *FRT*.

The **fnabs** instruction has two syntax forms. Each syntax form has a different effect on Condition Register Field 1.

Syntax form	Floating Point Status and Control Register	Record bit (Rc)	Condition Register Field 1
fnabs	None	0	None
fnabs.	None	1	FX,FEX,VX,OX

The two syntax forms of the **fnabs** instruction never affect the Floating Point Status and Control Register. If the syntax form sets the Record (Rc) bit to 1, the instruction affects the Floating Point Exception (FX), Floating Point Enabled Exception (FEX), Floating Point Invalid Operation Exception (VX), and Floating Point Overflow Exception (OX) bits in Condition Register Field 1.

Parameters

FRT Specifies target Floating Point register for operation.

FRB Specifies source Floating Point register for operation.

Examples

1. To negate the absolute contents of FPR 5 and place the result into FPR 6:

```
# Assume FPR 5 contains 0x400C 0000 0000 0000.
fnabs 6,5
# FPR 6 now contains 0xC00C 0000 0000 0000.
```

2. To negate the absolute contents of FPR 4, place the result into FPR 6, and set Condition Register Field 1 to reflect the result of the operation:

```
# Assume FPR 4 contains 0xC053 4000 0000 0000.
# Assume CR = 0.
fnabs. 6,4
# FPR 6 now contains 0xC053 4000 0000 0000.
# CR now contains 0x0.
```

Implementation Specifics

This instruction is part of Application Development Toolkit in AIX Base.

Related Information

Understanding Floating Point Move Instructions on page 1–13.

Understanding The Floating Point Status and Control Register on page 1–12.

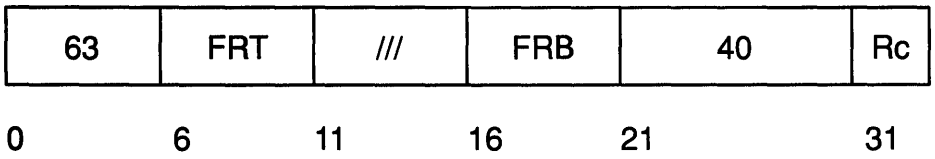
fneg (Floating Negate) Instruction

Purpose

Negates the contents of a Floating Point register and places the result into a Floating Point register.

Syntax

fneg *FRT,FRB*
fneg. *FRT,FRB*



Description

The **fneg** instruction places the negated contents of Floating Point Register *FRB* into the target Floating Point Register *FRT*.

The **fneg** instruction has two syntax forms. Each syntax form has a different effect on Condition Register Field 1.

Syntax form	Floating Point Status and Control Register	Record bit (Rc)	Condition Register Field 1
fneg	None	0	None
fneg.	None	1	FX,FEX,VX,OX

The two syntax forms of the **fneg** instruction never affect the Floating Point Status and Control Register. If the syntax form sets the Record (Rc) bit to 1, the instruction affects the Floating Point Exception (FX), Floating Point Enabled Exception (FEX), Floating Point Invalid Operation Exception (VX), and Floating Point Overflow Exception (OX) bits in Condition Register Field 1.

Parameters

FRT Specifies target Floating Point register for operation.

FRB Specifies source Floating Point register for operation.

Examples

1. To negate the contents of FPR 5 and place the result into FPR 6:

```
# Assume FPR 5 contains 0x400C 0000 0000 0000.
fneg 6,5
# FPR 6 now contains 0xC00C 0000 0000 0000.
```

2. To negate the contents of FPR 4, place the result into FPR 6, and set Condition Register Field 1 to reflect the result of the operation:

```
# Assume FPR 4 contains 0xC053 4000 0000 0000.
fneg. 6,4
# FPR 6 now contains 0x4053 4000 0000 0000.
# CR now contains 0x0000 0000.
```

Implementation Specifics

This instruction is part of Application Development Toolkit in AIX Base.

Related Information

Understanding Floating Point Move Instructions on page 1–13.

Understanding The Floating Point Status and Control Register on page 1–12.

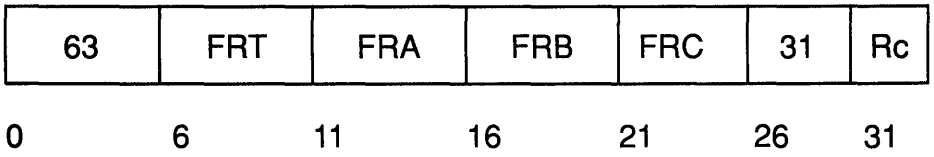
fnma (Floating Negative Multiply Add) Instruction

Purpose

Multiplies two 64-bit double precision floating point operands, adds the result to one 64-bit double precision floating point operand, and places the negative of the result in a Floating Point register.

Syntax

fnma *FRT,FRA,FRC,FRB*
fnma. *FRT,FRA,FRC,FRB*



Description

The **fnma** instruction multiplies the 64-bit double precision floating point operand in Floating Point Register *FRA* by the 64-bit double precision floating point operand in Floating Point Register *FRC*, and adds the 64-bit double precision floating point operand in Floating Point Register *FRB* to the result of the multiplication.

The result of the addition is rounded under control of the Floating Point Rounding Control Field *RN* of the Floating Point Status and Control Register.

- If an operand is a denormalized number, then it is prenormalized before the operation is begun.

This instruction is identical to the **fma** (Floating Multiply Add) with the final result negated, but with the following exceptions:

- QNaNs propagate with no effect on their "sign" bit.
- QNaNs that are generated as the result of a disabled Invalid Operation Exception have a "sign" bit of zero.
- SNaNs that are converted to QNaNs as the result of a disabled Invalid Operation Exception have no effect on their "sign" bit.

The Floating Point Result Flags field of the Floating Point Status and Control Register is set to the class and sign of the result except for Invalid Operation Exceptions when the Floating Point Invalid Operation Exception Enable bit is 1.

The **fnma** instruction has two syntax forms. Each syntax form has a different effect on Condition Register Field 1.

Syntax form	Floating Point Status and Control Register	Record bit (Rc)	Condition Register Field 1
fnma	C,FL,FG,FE,FU,FR,FI,OX,UX, XX,VXSNAN,VXISI,VXIMZ	0	None
fnma.	C,FL,FG,FE,FU,FR,FI,OX,UX, XX,VXSNAN,VXISI,VXIMZ	1	FX,FEX,VX,OX

The two syntax forms of the **fnma** instruction always affect the Floating Point Status and Control Register. If the syntax form sets the Record (Rc) bit to 1, the instruction affects the Floating Point Exception (FX), Floating Point Enabled Exception (FEX), Floating Point Invalid Operation Exception (VX), and Floating Point Overflow Exception (OX) bits in Condition Register Field 1.

Note: Rounding occurs before the result of the addition is negated. Depending on *RN*, an inexact value may result.

Parameters

FRT	Specifies target Floating Point register for operation.
FRA	Specifies source Floating Point register for operation.
FRB	Specifies source Floating Point register for operation.
FRC	Specifies source Floating Point register for operation.

Examples

1. To multiply the contents of FPR 4 and FPR 5, add the result to the contents of FPR 7, store the negated result in FPR 6, and set the Floating Point Status and Control Register to reflect the result of the operation:

```
# Assume FPR 4 contains 0xC053 4000 0000 0000.
# Assume FPR 5 contains 0x400C 0000 0000 0000.
# Assume FPR 7 contains 0x3DE2 6AB4 B33c 110A.
# Assume RM = 0 and FPSCR = 0.
fnma 6,4,5,7
# FPR 6 now contains 0x4070 D7FF FFFF F6CB.
# FPSCR now contains 0x8206 4000.
```

2. To multiply the contents of FPR 4 and FPR 5, add the result to the contents of FPR 7, store the negated result in FPR 6, and set the Floating Point Status and Control Register and Condition Register Field 1 to reflect the result of the operation:

```
# Assume FPR 4 contains 0xC053 4000 0000 0000.
# Assume FPR 5 contains 0x400C 0000 0000 0000.
# Assume FPR 7 contains 0x3DE2 6AB4 B33c 110A.
# Assume RM = 0, FPSCR = 0, and CR = 0.
fnma. 6,4,5,7
# FPR 6 now contains 0x4070 D7FF FFFF F6CB.
# FPSCR now contains 0x8206 4000.
# CR now contains 0x0800 0000.
```

Implementation Specifics

This instruction is part of Application Development Toolkit in AIX Base.

Related Information

Understanding Floating Point Accumulate Instructions on page 1–14.

Understanding The Floating Point Status and Control Register on page 1–12.

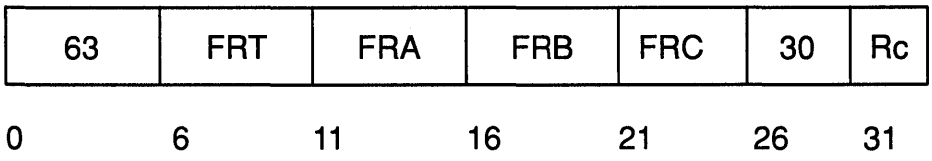
fnms (Floating Negative Multiply Subtract) Instruction

Purpose

Multiplies two 64-bit double precision floating point operands, subtracts one 64-bit double precision floating point operand from the result, and places the negative of the result in a Floating Point register.

Syntax

fnms *FRT,FRA,FRC,FRB*
fnms. *FRT,FRA,FRC,FRB*



Description

The **fnms** instruction multiplies the 64-bit double precision floating point operand in Floating Point Register *FRA* by the 64-bit double precision floating point operand in Floating Point Register *FRC*, subtracts the 64-bit double precision floating point operand in Floating Point Register *FRB* from the result of the multiplication, and places the negated result in the target Floating Point Register *FRT*.

The subtraction result is rounded under control of the Floating Point Rounding Control Field *RN* of the Floating Point Status and Control Register.

- If an operand is a denormalized number, then it is prenormalized before the operation is begun.

This instruction is identical to the **fms** (Floating Multiply Subtract) with the final result negated, but with the following exceptions:

- QNaNs propagate with no effect on their "sign" bit.
- QNaNs that are generated as the result of a disabled Invalid Operation Exception have a "sign" bit of zero.
- SNaNs that are converted to QNaNs as the result of a disabled Invalid Operation Exception have no effect on their "sign" bit.

The Floating Point Result Flags field of the Floating Point Status and Control Register is set to the class and sign of the result except for Invalid Operation Exceptions when the Floating Point Invalid Operation Exception Enable bit is 1.

The **fnms** instruction has two syntax forms. Each syntax form has a different effect on Condition Register Field 1.

Syntax form	Floating Point Status and Control Register	Record bit (Rc)	Condition Register Field 1
fnms	C,FL,FG,FE,FU,FR,FI,OX,UX, XX,VXSNAN,VXISI,VXIMZ	0	None
fnms.	C,FL,FG,FE,FU,FR,FI,OX,UX, XX,VXSNAN,VXISI,VXIMZ	1	FX,FEX,VX,OX

The two syntax forms of the **fnms** instruction always affect the Floating Point Status and Control Register. If the syntax form sets the Record (Rc) bit to 1, the instruction affects the Floating Point Exception (FX), Floating Point Enabled Exception (FEX), Floating Point Invalid Operation Exception (VX), and Floating Point Overflow Exception (OX) bits in Condition Register Field 1.

Note: Rounding occurs before the result of the addition is negated. Depending on *RN*, an inexact value may result.

Parameters

FRT Specifies target Floating Point register for operation.

FRA Specifies first source Floating Point register for operation.

FRB Specifies second source Floating Point register for operation.

FRC Specifies third source Floating Point register for operation.

Examples

1. To multiply the contents of FPR 4 and FPR 5, subtract the contents of FPR 7 from the result, store the negated result in FPR 6, and set the Floating Point Status and Control Register and Condition Register Field 1 to reflect the result of the operation:

```
# Assume FPR 4 contains 0xC053 4000 0000 0000.
# Assume FPR 5 contains 0x400C 0000 0000 0000.
# Assume FPR 7 contains 0x3DE2 6AB4 B33c 110A.
# Assume RM = 0 and FPSCR = 0.
fnms 6,4,5,7
# FPR 6 now contains 0x4070 D800 0000 0935.
# FPSCR now contains 0x8202 4000.
```

2. To multiply the contents of FPR 4 and FPR 5, subtract the contents of FPR 7 from the result, store the negated result in FPR 6, and set the Floating Point Status and Control Register and Condition Register Field 1 to reflect the result of the operation:

```
# Assume FPR 4 contains 0xC053 4000 0000 0000.
# Assume FPR 5 contains 0x400C 0000 0000 0000.
# Assume FPR 7 contains 0x3DE2 6AB4 B33c 110A.
# Assume RM = 0, FPSCR = 0, and CR = 0.
fnms. 6,4,5,7
# FPR 6 now contains 0x4070 D800 0000 0935.
# FPSCR now contains 0x8202 4000.
# CR now contains 0x0800 0000.
```

Implementation Specifics

This instruction is part of Application Development Toolkit in AIX Base.

Related Information

Understanding Floating Point Accumulate Instructions on page 1–14.

Understanding The Floating Point Status and Control Register on page 1–12.

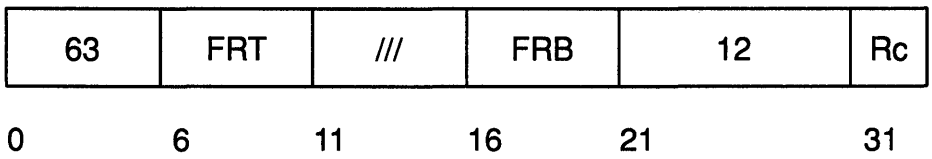
frsp (Floating Round to Single Precision) Instruction

Purpose

Rounds a 64-bit double precision floating point operand to single precision and places the result in a Floating Point register.

Syntax

frsp *FRT,FRB*
frsp. *FRT,FRB*



Description

The **frsp** instruction rounds the 64-bit double precision floating point operand in Floating Point Register *FRB* to single precision using the rounding mode specified by the Floating Rounding Control field of the Floating Point Status and Control Register and places the result in the target Floating Point Register *FRT*.

The Floating Point Result Flags field of the Floating Point Status and Control Register is set to the class and sign of the result except for Invalid Operation (SNaN) when Floating Point Status and Control Register Floating Point Invalid Operation Exception Enable bit is 1.

The **frsp** instruction has two syntax forms. Each syntax form has a different effect on Condition Register Field 1.

Syntax form	Floating Point Status and Control Register	Record bit (Rc)	Condition Register Field 1
frsp	C,FL,FG,FE,FU,FR,FI,OX,UX, XX,VXSNAN	0	None
frsp.	C,FL,FG,FE,FU,FR,FI,OX,UX, XX,VXSNAN	1	FX,FEX,VX,OX

The two syntax forms of the **frsp** instruction always affect the Floating Point Status and Control Register. If the syntax form sets the Record (Rc) bit to 1, the instruction affects the Floating Point Exception (FX), Floating Point Enabled Exception (FEX), Floating Point Invalid Operation Exception (VX), and Floating Point Overflow Exception (OX) bits in Condition Register Field 1.

Note: The **frsp** instruction may produce incorrect results when all of the following conditions are met.

1. The **frsp** instruction uses the target register of a previous floating point arithmetic operation as its source register (*FRB*). The **frsp** instruction is said to be *dependent* on the preceding floating point arithmetic operation when it uses this register for source.
2. Less than two nondependent floating point arithmetic operations occur between the **frsp** instruction and the operation on which it is dependent.
3. The magnitude of the double precision result of the arithmetic operation is less than 2**128 before rounding.
4. The magnitude of the double precision result after rounding is exactly 2**128.

Error Result

If the error occurs, the magnitude of the result placed in the target register *FRT* is 2**128:

X'47F0000000000000' or X'C7F0000000000000'

This is not a valid single precision value. The setting of the Floating Point Status and Control Register and Condition Register will be the same as if the result does not overflow.

Avoiding Errors

If the above error will cause significant problems in an application, either of the following two methods can be used to avoid the error.

1. Place two nondependent floating point operations between a floating point arithmetic operation and the dependent **frsp** instruction.The target registers for these nondependent floating point operations should not be the same register that the **frsp** instruction uses as source register *FRB*.
2. Insert two **frsp** operations when the **frsp** instruction may be dependent on an arithmetic operation that precedes it by less than three floating point instructions.

Either solution will degrade performance by an amount dependent on the particular application.

Parameters

FRT	Specifies target Floating Point register for operation.
FRB	Specifies source Floating Point register for operation.

Examples

1. To round the contents of FPR 4 to single precision, place the result in a FPR 6, and set the Floating Point Status and Control Register to reflect the result of the operation:

```
# Assume FPR 4 contains 0xC053 4000 0000 0000.
# Assume FPSCR = 0.
frsp 6,4
# FPR 6 now contains 0xC053 4000 0000 0000.
# FPSCR now contains 0x0000 8000.
```

2. To round the contents of FPR 4 to single precision, place the result in a FPR 6, and set the Floating Point Status and Control Register and Condition Register Field 1 to reflect the result of the operation:

```
# Assume FPR 4 contains 0xFFFF FFFF FFFF FFFF.
# Assume FPSCR = 0.
frsp. 6,4
# FPR 6 now contains 0xFFFF FFFF EC00 0000.
# FPSCR now contains 0x0001 1000.
# CR now contains 0x0000 0000.
```

Implementation Specifics

This instruction is part of Application Development Toolkit in AIX Base.

Related Information

Understanding Floating Point Arithmetic Instructions on page 1–14.

Understanding The Floating Point Status and Control Register on page 1–12.

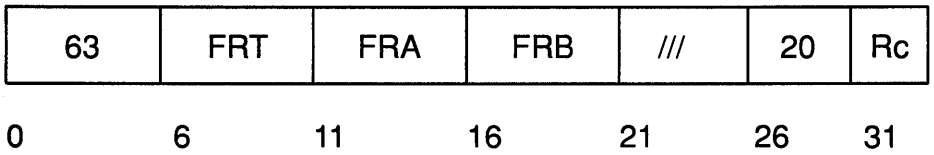
fs (Floating Subtract) Instruction

Purpose

Subtracts one 64-bit double precision floating point operand from another and places the result in a Floating Point register.

Syntax

fs *FRT,FRA,FRB*
fs. *FRT,FRA,FRB*



Description

The **fs** instruction subtracts the 64-bit double precision floating point operand in Floating Point Register *FRB* from the 64-bit double precision floating point operand in Floating Point Register *FRA*. The result is rounded under control of the Floating Point Rounding Control Field *RN* of the Floating Point Status and Control Register and is placed in the target Floating Point register *FRT*.

The execution of the **fs** instruction is identical to that of **fa**, except that the contents of Floating Point Register *FRB* participate in the operation with bit 0 inverted.

The Floating Point Result Flags field of the Floating Point Status and Control Register is set to the class and sign of the result except for Invalid Operation Exceptions when the Floating Point Invalid Operation Exception Enable bit is 1.

The **fs** instruction has two syntax forms. Each syntax form has a different effect on Condition Register Field 1.

Syntax form	Floating Point Status and Control Register	Record bit (Rc)	Condition Register Field 1
fs	C,FL,FG,FE,FU,FR,FI,OX,UX, XX,VXSNAN,VXISI	0	None
fs.	C,FL,FG,FE,FU,FR,FI,OX,UX, XX,VXSNAN,VXISI	1	FX,FEX,VX,OX

The two syntax forms of the **fs** instruction always affect the Floating Point Status and Control Register. If the syntax form sets the Record (Rc) bit to 1, the instruction affects the Floating Point Exception (FX), Floating Point Enabled Exception (FEX), Floating Point Invalid Operation Exception (VX), and Floating Point Overflow Exception (OX) bits in Condition Register Field 1.

Parameters

FRT Specifies target Floating Point register for operation.

FRA Specifies source Floating Point register for operation.

FRB Specifies source Floating Point register for operation.

Examples

1. To subtract the contents of FPR 5 from the contents of FPR 4, place the result in FPR 6, and set the Floating Point Status and Control Register to reflect the result of the operation:

```
# Assume FPR 4 contains 0xC053 4000 0000 0000.
# Assume FPR 5 contains 0x400C 0000 0000 0000.
# Assume RM = 0 and FPSCR = 0.
fs 6,4,5
# FPR 6 now contains 0xC054 2000 0000 0000.
# FPSCR now contains 0x0000 8000.
```

2. To subtract the contents of FPR 5 from the contents of FPR 4, place the result in FPR 6, and set the Floating Point Status and Control Register and Condition Register Field 1 to reflect the result of the operation:

```
# Assume FPR 4 contains 0xC053 4000 0000 0000.
# Assume FPR 5 contains 0x400C 0000 0000 0000.
# Assume RM = 0, FPSCR = 0, and CR = 0.
fs. 6,5,4
# FPR 6 now contains 0x4054 2000 0000 0000.
# FPSCR now contains 0x0000 4000.
# CR now contains 0x0000 0000.
```

Implementation Specifics

This instruction is part of Application Development Toolkit in AIX Base.

Related Information

Understanding Floating Point Arithmetic Instructions on page 1–14.

Understanding The Floating Point Status and Control Register on page 1–12.

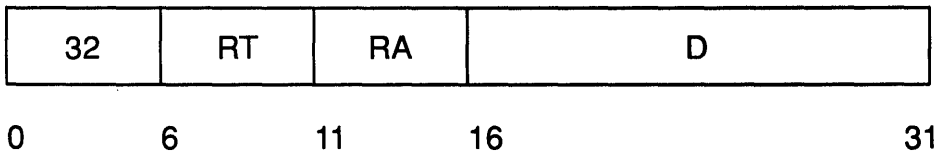
l (Load) Instruction

Purpose

Loads a word of data from a specified location in memory into a general purpose register.

Syntax

l *RT,D(RA)*



Description

The l instruction loads a word in storage from a specified location in memory addressed by the effective address (EA) into the target General Purpose Register *RT*.

The effective address (EA) is the sum of the contents of General Purpose Register *RA* and *D*, a 16-bit signed two's complement integer sign extended to 32 bits, if *RA* is not 0. If *RA* is 0, then the effective address (EA) is *D*.

- If alignment checking is disabled, i.e., the alignment bit (AL) in the Machine Status Register (MSR) is 0, then the two low order bits of the effective address are ignored.
- If alignment checking is enabled, and the two low order bits are not b'00', then the hardware attempts to perform the unaligned storage access. If the hardware cannot perform the unaligned storage access, an Alignment Interrupt is generated.

The l instruction has one syntax form and does not affect the Fixed Point Exception Register or Condition Register Field 0.

Parameters

RT	Specifies target general purpose register where result of operation is stored.
D	16-bit signed two's complement integer sign extended to 32 bits for EA calculation.
RA	Specifies source general purpose register for EA calculation.

Examples

1. To load a word from memory into GPR 6:

```
.csect data[rw]
# Assume GPR 5 contains address of csect data[rw].
storage: .long 0x4
.csect text[pr]
l 6,storage(5)
# GPR 6 now contains 0x0000 0004.
```

Implementation Specifics

This instruction is part of Application Development Toolkit in AIX Base.

Related Information

Understanding Fixed Point Load Instructions on page 1–6.

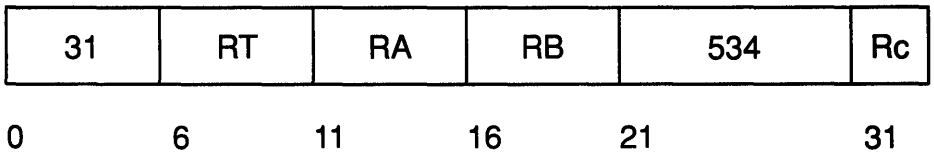
lbrx (Load Byte Reverse Indexed) Instruction

Purpose

Loads a byte–reversed word of data from a specified location in memory into a general purpose register.

Syntax

lbrx *RT,RA,RB*



Description

The **lbrx** instruction loads a byte–reversed word in storage from a specified location in memory addressed by the effective address (EA) into the target General Purpose Register *RT*.

- Bits 00–07 of the word in storage addressed by EA are stored into bits 24–31 of GPR *RT*.
- Bits 08–15 of the word in storage addressed by EA are stored into bits 16–23 of GPR *RT*.
- Bits 16–23 of the word in storage addressed by EA are stored into bits 08–15 of GPR *RT*.
- Bits 24–31 of the word in storage addressed by EA are stored into bits 00–07 of GPR *RT*.

The effective address (EA) is the sum of the contents of General Purpose Register *RA* and General Purpose Register *RB* if *RA* is not 0. If *RA* is 0, then the effective address (EA) is the contents of *RB*.

- If alignment checking is disabled, i.e., the alignment bit (AL) in the Machine Status Register (MSR) is 0, then the two low order bits of the effective address are ignored.
- If alignment checking is enabled, and the two low order bits are not b'00', then the hardware attempts to perform the unaligned storage access. If the hardware cannot perform the unaligned storage access, an Alignment Interrupt is generated.

The **lbrx** instruction has one syntax form and does not affect the Fixed Point Exception Register or Condition Register Field 0.

Parameters

RT	Specifies target general purpose register where result of operation is stored.
RA	Specifies source general purpose register for EA calculation.
RB	Specifies source general purpose register for EA calculation.

Examples

1. To load a byte–reversed word from memory into GPR 6:

```
storage:  .long 0x0000 ffff
 .
 .
# Assume GPR 4 contains 0x0000 0000.
# Assume GPR 5 contains address of storage.
lbrx 6,4,5
# GPR 6 now contains 0xffff 0000.
```

Implementation Specifics

This instruction is part of Application Development Toolkit in AIX Base.

Related Information

Understanding Fixed Point Load Instructions on page 1–6.

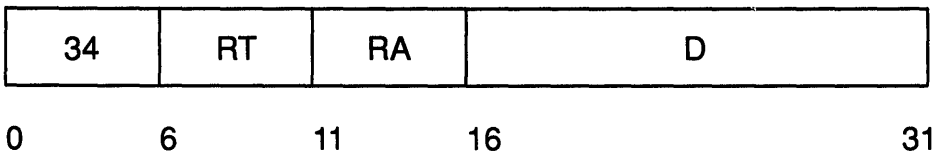
lbz (Load Byte And Zero) Instruction

Purpose

Loads a byte of data from a specified location in memory into a general purpose register and sets the remaining 24 bits to 0.

Syntax

lbz *RT,D(RA)*



Description

The **lbz** instruction loads a byte in storage addressed by EA into bits 24–31 of the target General Purpose Register *RT* and sets bits 0–23 of General Purpose Register *RT* to 0.

The effective address (EA) is the sum of the contents of General Purpose Register *RA* and *D*, a 16-bit signed two's complement integer sign extended to 32 bits, if *RA* is not 0. If *RA* is 0, then the effective address (EA) is *D*.

The **lbz** instruction has one syntax form and does not affect the Fixed Point Exception Register or Condition Register Field 0.

Parameters

RT	Specifies target general purpose register where result of operation is stored.
D	16-bit signed two's complement integer sign extended to 32 bits for EA calculation.
RA	Specifies source general purpose register for EA calculation.

Examples

1. To load a byte of data from a specified location in memory into GPR 6 and set the remaining 24 bits to 0:

```
.csect data[rw]
storage:  .byte 'a
# Assume GPR 5 contains the address of csect data[rw].
.csect text[pr]
lbz 6,storage(5)
# GPR 6 now contains 0x0000 0061.
```

Implementation Specifics

This instruction is part of Application Development Toolkit in AIX Base.

Related Information

Understanding Fixed Point Load Instructions on page 1–6.

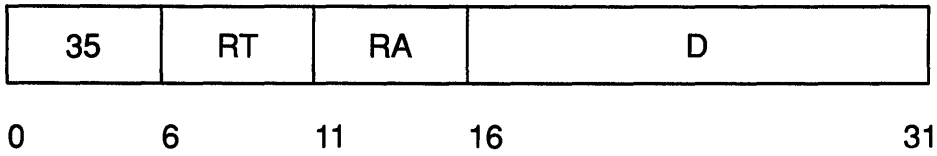
lbzu (Load Byte And Zero With Update) Instruction

Purpose

Loads a byte of data from a specified location in memory into a general purpose register, sets the remaining 24 bits to 0, and possibly places the address in the a second general purpose register.

Syntax

lbzu *RT,D(RA)*



Description

The **lbzu** instruction loads a byte in storage addressed by EA into bits 24–31 of the target General Purpose Register *RT* and sets bits 0–23 of General Purpose Register *RT* to 0.

The effective address (EA) is the sum of the contents of General Purpose Register *RA* and *D*, a 16-bit signed two's complement integer sign extended to 32 bits, if *RA* is not 0. If *RA* is 0, then the effective address (EA) is *D*.

- If *RA* does not equal *RT* and *RA* does not equal 0 and the storage access does not cause Alignment Interrupt or a Data Storage Interrupt, then the effective address is stored in General Purpose Register *RA*.

The **lbzu** instruction has one syntax form and does not affect the Fixed Point Exception Register or Condition Register Field 0.

Parameters

RT	Specifies target general purpose register where result of operation is stored.
D	16-bit signed two's complement integer sign extended to 32 bits for EA calculation.
RA	Specifies source general purpose register for EA calculation and possible address update.

Examples

1. To load a byte of data from a specified location in memory into GPR 6, set the remaining 24 bits to 0, and place the address in GPR 5:

```
.csect data[rw]
storage:  .byte 0x61
# Assume GPR 5 contains the address of csect data[rw].
.csect text[pr]
lbzu 6,storage(5)
# GPR 6 now contains 0x0000 0061.
# GPR 5 now contains the storage address.
```

Implementation Specifics

This instruction is part of Application Development Toolkit in AIX Base.

Related Information

Understanding Fixed Point Load with Update Instructions on page 1–6.

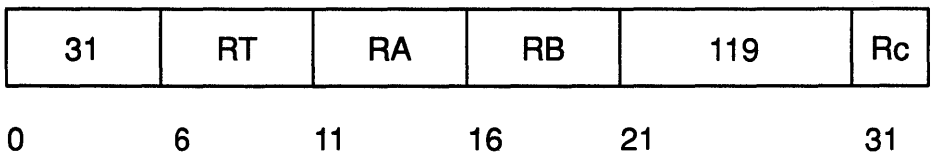
lbzux (Load Byte And Zero With Update Indexed) Instruction

Purpose

Loads a byte of data from a specified location in memory into a general purpose register, setting the remaining 24 bits to 0, and places the address in the a second general purpose register.

Syntax

lbzux *RT,RA,RB*



Description

The **lbzux** instruction loads a byte in storage addressed by the effective address (EA) into bits 24–31 of the target General Purpose Register *RT* and sets bits 0–23 of General Purpose Register *RT* to 0.

The effective address (EA) is the sum of the contents of General Purpose Register *RA* and General Purpose Register *RB* if *RA* is not 0. If *RA* is 0, then the effective address (EA) is the contents of *RB*.

- If *RA* does not equal *RT* and *RA* does not equal 0 and the storage access does not cause Alignment Interrupt or a Data Storage Interrupt, then the effective address is stored in General Purpose Register *RA*.

The **lbzux** instruction has one syntax form and does not affect the Fixed Point Exception Register.

Parameters

RT	Specifies target general purpose register where result of operation is stored.
RA	Specifies source general purpose register for EA calculation and possible address update.
RB	Specifies source general purpose register for EA calculation.

Examples

1. To load the value located at *storage* into GPR 6 and load the address of *storage* into GPR 5:

```
storage: .byte 0x40
   .
   .
# Assume GPR 5 contains 0x0000 0000.
# Assume GPR 4 is the storage address.
lbzux 6,5,4
# GPR 6 now contains 0x0000 0040.
# GPR 5 now contains the storage address.
```

Implementation Specifics

This instruction is part of Application Development Toolkit in AIX Base.

Related Information

Understanding Fixed Point Load with Update Instructions on page 1–6.

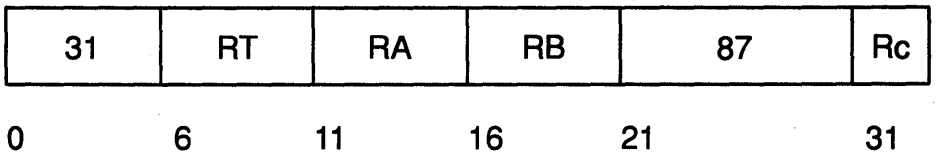
lbzx (Load Byte And Zero Indexed) Instruction

Purpose

Loads a byte of data from a specified location in memory into a general purpose register and sets the remaining 24 bits to 0.

Syntax

lbzx *RT,RA,RB*



Description

The **lbzx** instruction loads a byte in storage addressed by the effective address (EA) into bits 24–31 of the target General Purpose Register *RT* and sets bits 0–23 of General Purpose Register *RT* to 0.

The effective address (EA) is the sum of the contents of General Purpose Register *RA* and General Purpose Register *RB* if *RA* is not 0. If *RA* is 0, then the effective address (EA) is *D*.

The **lbzx** instruction has one syntax form and does not affect the Fixed Point Exception Register.

Parameters

RT	Specifies target general purpose register where result of operation is stored.
RA	Specifies source general purpose register for EA calculation.
RB	Specifies source general purpose register for EA calculation.

Examples

1. To load the value located at *storage* into GPR 6:

```
storage:  .byte 0x61
   .
   .
# Assume GPR 5 contains 0x0000 0000.
# Assume GPR 4 is the storage address.
lbzx 6,5,4
# GPR 6 now contains 0x0000 0061.
```

Implementation Specifics

This instruction is part of Application Development Toolkit in AIX Base.

Related Information

Understanding Fixed Point Load Instructions on page 1–6.

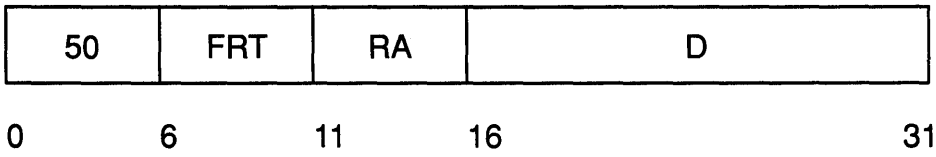
lfd (Load Floating Point Double) Instruction

Purpose

Loads a doubleword of data from a specified location in memory into a floating point register.

Syntax

lfd *FRT,D(RA)*



Description

The **lfd** instruction loads a doubleword in storage from a specified location in memory addressed by the effective address (EA) into the target Floating Point Register *FRT*.

The effective address (EA) is the sum of the contents of General Purpose Register *RA* and *D*, a 16-bit signed two's complement integer sign extended to 32 bits, if *RA* is not 0. If *RA* is 0, then the effective address (EA) is *D*.

- If alignment checking is disabled, i.e., the alignment bit (AL) in the Machine Status Register (MSR) is 0, then the three low order bits of the effective address are ignored.
- If alignment checking is enabled, and the three low order bits are not b'000', then the hardware attempts to perform the unaligned storage access. If the hardware cannot perform the unaligned storage access, an Alignment Interrupt is generated.

The **lfd** instruction has one syntax form and does not affect the Fixed Point Exception Register or Condition Register Field 0.

Parameters

FRT	Specifies target general purpose register where result of operation is stored.
D	16-bit signed two's complement integer sign extended to 32 bits for EA calculation.
RA	Specifies source general purpose register for EA calculation.

Examples

1. To load a doubleword from memory into FPR 6:

```
.csect data[rw]
storage: .double 0x1
# Assume GPR 5 contains the address of csect data[rw].
.csect text[pr]
lfd 6,storage(5)
# FPR 6 now contains 0x3FF0 0000 0000 0000.
```

Implementation Specifics

This instruction is part of Application Development Toolkit in AIX Base.

Related Information

Understanding Floating Point Load Instructions on page 1–13.

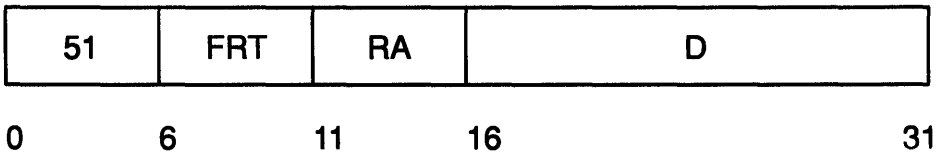
lfdu (Load Floating Point Double With Update) Instruction

Purpose

Loads a doubleword of data from a specified location in memory into a floating point register and possibly places the specified address in a general purpose register.

Syntax

lfdu *FRT,D(RA)*



Description

The **lfdu** instruction loads a doubleword in storage from a specified location in memory addressed by the effective address (EA) into the target Floating Point Register *FRT*.

The effective address (EA) is the sum of the contents of General Purpose Register *RA* and *D*, a 16-bit signed two's complement integer sign extended to 32 bits, if *RA* is not 0. If *RA* is 0, then the effective address (EA) is *D*.

- If *RA* does not equal 0 and the storage access does not cause Alignment Interrupt or a Data Storage Interrupt, then the effective address is stored in General Purpose Register RA.
- If alignment checking is disabled, i.e., the alignment bit (AL) in the Machine Status Register (MSR) is 0, then the three low order bits of the effective address are ignored.
- If alignment checking is enabled, and the three low order bits are not b'000', then the hardware attempts to perform the unaligned storage access. If the hardware cannot perform the unaligned storage access, an Alignment Interrupt is generated.

The **lfdu** instruction has one syntax form and does not affect the Fixed Point Exception Register or Condition Register Field 0.

Parameters

FRT	Specifies target general purpose register where result of operation is stored.
D	16-bit signed two's complement integer sign extended to 32 bits for EA calculation.
RA	Specifies source general purpose register for EA calculation and possible address update.

Examples

1. To load a doubleword from memory into FPR 6 and store the address in GPR 5:

```
.csect data[rw]
storage: .double 0x1
# Assume GPR 5 contains the address of csect data[rw].
.csect text[pr]
lfdu 6,storage(5)
# FPR 6 now contains 0x3FF0 0000 0000 0000.
# GPR 5 now contains the storage address.
```

Implementation Specifics

This instruction is part of Application Development Toolkit in AIX Base.

Related Information

Understanding Floating Point Load Instructions on page 1–13.

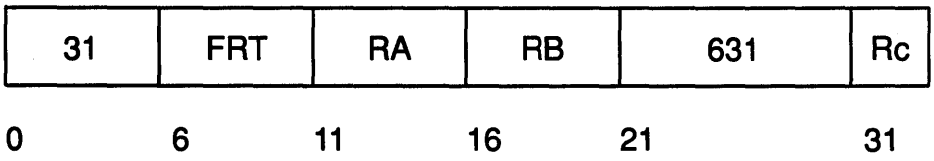
lfdux (Load Floating Point Double With Update Indexed) Instruction

Purpose

Loads a doubleword of data from a specified location in memory into a floating point register and possibly places the specified address in a general purpose register.

Syntax

lfdux *FRT,RA,RB*



Description

The **lfdux** instruction loads a doubleword in storage from a specified location in memory addressed by the effective address (EA) into the target Floating Point Register *FRT*.

The effective address (EA) is the sum of the contents of General Purpose Register *RA* and General Purpose Register *RB* if *RA* is not 0. If *RA* is 0, then the effective address (EA) is *D*.

- If *RA* does not equal 0 and the storage access does not cause Alignment Interrupt or a Data Storage Interrupt, then the effective address is stored in General Purpose Register RA.
- If alignment checking is disabled, i.e., the alignment bit (AL) in the Machine Status Register (MSR) is 0, then the three low order bits of the effective address are ignored.
- If alignment checking is enabled, and the three low order bits are not b'000', then the hardware attempts to perform the unaligned storage access. If the hardware cannot perform the unaligned storage access, an Alignment Interrupt is generated.

The **lfdux** instruction has one syntax form and does not affect the Fixed Point Exception Register.

Parameters

FRT	Specifies target general purpose register where result of operation is stored.
RA	Specifies source general purpose register for EA calculation.
RB	Specifies source general purpose register for EA calculation.

Examples

1. To load a doubleword from memory into FPR 6 and store the address in GPR 5:

```
.csect data[rw]
storage: .double 0x1
# Assume GPR 5 contains the address of csect data[rw].
# Assume GPR 4 contains the displacement of storage relative
# to .csect data[rw].
.csect text[pr]
lfdux 6,5,4
# FPR 6 now contains 0x3FF0 0000 0000 0000.
# GPR 5 now contains the storage address.
```

Implementation Specifics

This instruction is part of Application Development Toolkit in AIX Base.

Related Information

Understanding Floating Point Load Instructions on page 1–13.

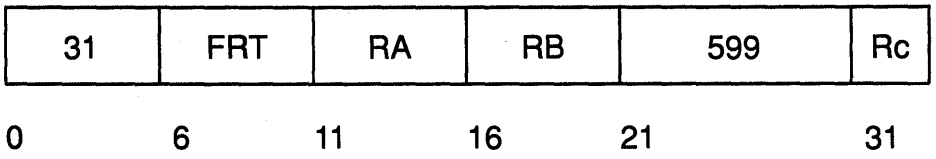
lfdx (Load Floating Point Double Indexed) Instruction

Purpose

Loads a doubleword of data from a specified location in memory into a floating point register.

Syntax

lfdx *FRT,RA,RB*



Description

The **lfdx** instruction loads a doubleword in storage from a specified location in memory addressed by the effective address (EA) into the target Floating Point Register *FRT*.

The effective address (EA) is the sum of the contents of General Purpose Register *RA* and General Purpose Register *RB* if *RA* is not 0. If *RA* is 0, then the effective address (EA) is the contents of *RB*.

- If alignment checking is disabled, i.e., the alignment bit (AL) in the Machine Status Register (MSR) is 0, then the three low order bits of the effective address are ignored.
- If alignment checking is enabled, and the three low order bits are not b'000', then the hardware attempts to perform the unaligned storage access. If the hardware cannot perform the unaligned storage access, an Alignment Interrupt is generated.

The **lfdx** instruction has one syntax form and does not affect the Fixed Point Exception Register or Condition Register Field 0.

Parameters

FRT	Specifies target floating point register where data is stored.
RA	Specifies source general purpose register for EA calculation.
RB	Specifies source general purpose register for EA calculation.

Examples

1. To load a doubleword from memory into FPR 6:

```
storage: .double 0x1
   .
   .
# Assume GPR 4 contains the storage address.
lfdx 6,0,4
# FPR 6 now contains 0x3FF0 0000 0000 0000.
```

Implementation Specifics

This instruction is part of Application Development Toolkit in AIX Base.

Related Information

Understanding Floating Point Load Instructions on page 1–13.

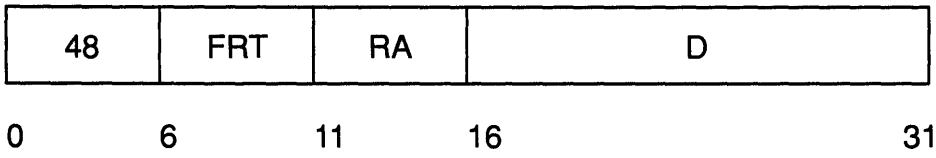
lfs (Load Floating Point Single) Instruction

Purpose

Loads a floating point single precision number which is converted into a floating point double precision number into a floating point register.

Syntax

lfs *FRT,D(RA)*



Description

The **lfs** instruction converts a floating point single precision word in storage addressed by the effective address (EA) to floating point double precision and loads the result into Floating Point Register *FRT*.

The effective address (EA) is the sum of the contents of General Purpose Register *RA* and *D*, a 16-bit signed two's complement integer sign extended to 32 bits, if *RA* is not 0. If *RA* is 0, then the effective address (EA) is *D*.

- If alignment checking is disabled, i.e., the alignment bit (AL) in the Machine Status Register (MSR) is 0, then the two low order bits of the effective address are ignored.
- If alignment checking is enabled, and the two low order bits are not b'00', then the hardware attempts to perform the unaligned storage access. If the hardware cannot perform the unaligned storage access, an Alignment Interrupt is generated.

The **lfs** instruction has one syntax form and does not affect the Fixed Point Exception Register or Condition Register Field 0.

Parameters

FRT	Specifies target floating point register where data is stored.
D	16-bit signed two's complement integer sign extended to 32 bits for EA calculation.
RA	Specifies source general purpose register for EA calculation.

Examples

1. To load the single precision contents of storage into FPR 6:

```
.csect data[rw]
storage: .float 0x1
# Assume GPR 5 contains the address csect data[rw].
.csect text[pr]
lfs 6,storage(5)
# FPR 6 now contains 0x3FF0 0000 0000 0000.
```

Implementation Specifics

This instruction is part of Application Development Toolkit in AIX Base.

Related Information

Understanding Floating Point Load Instructions on page 1–13.

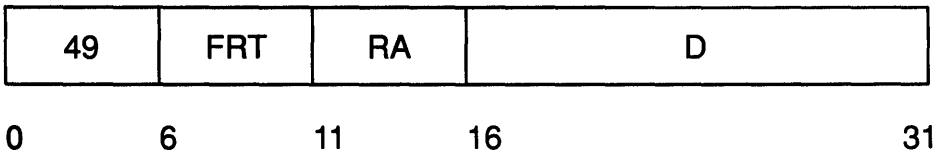
lfsu (Load Floating Point Single With Update) Instruction

Purpose

Loads a floating point single precision number which is converted into a floating point double precision number into a floating point register and possibly places the effective address in a general purpose register.

Syntax

lfsu *FRT,D(RA)*



Description

The **lfsu** instruction converts a floating point single precision word in storage addressed by the effective address (EA) to floating point double precision and loads the result into Floating Point Register *FRT*.

The effective address (EA) is the sum of the contents of General Purpose Register *RA* and *D*, a 16-bit signed two's complement integer sign extended to 32 bits, if *RA* is not 0. If *RA* is 0, then the effective address (EA) is *D*.

- If *RA* does not equal 0 and the storage access does not cause an Alignment Interrupt or a Data Storage Interrupt, then the effective address is stored in General Purpose Register RA.
- If alignment checking is disabled, i.e., the alignment bit (AL) in the Machine Status Register (MSR) is 0, then the two low order bits of the effective address are ignored.
- If alignment checking is enabled, and the two low order bits are not b'00', then the hardware attempts to perform the unaligned storage access. If the hardware cannot perform the unaligned storage access, an Alignment Interrupt is generated.

The **lfsu** instruction has one syntax form and does not affect the Fixed Point Exception Register or Condition Register Field 0.

Parameters

FRT	Specifies target floating point register where data is stored.
D	16-bit signed two's complement integer sign extended to 32 bits for EA calculation.
RA	Specifies source general purpose register for EA calculation and possible address update.

Examples

1. To load the single precision contents of storage, which is converted to double precision, into FPR 6 and store the effective address in GPR 5:

```
.csect data[rw]
storage: .float 0x1
.csect text[pr]
# Assume GPR 5 contains the storage address.
lfsu 6,0(5)
# FPR 6 now contains 0x3FF0 0000 0000 0000.
# GPR 5 now contains the storage address.
```

Implementation Specifics

This instruction is part of Application Development Toolkit in AIX Base.

Related Information

Understanding Floating Point Load Instructions on page 1–13.

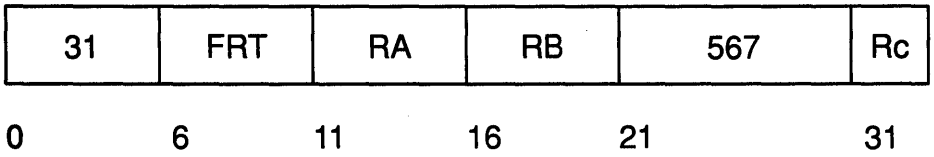
lfsux (Load Floating Point Single With Update Indexed) Instruction

Purpose

Loads a floating point single precision number which is converted into a floating point double precision number into a floating point register and possibly places the effective address in a general purpose register.

Syntax

lfsux *FRT,RA,RB*



Description

The **lfsux** instruction converts a floating point single precision word in storage addressed by the effective address (EA) to floating point double precision and loads the result into Floating Point Register *FRT*.

The effective address (EA) is the sum of the contents of General Purpose Register *RA* and General Purpose Register *RB* if *RA* is not 0. If *RA* is 0, then the effective address (EA) is the contents of *RB*.

- If *RA* does not equal 0 and the storage access does not cause an Alignment Interrupt or a Data Storage Interrupt, then the effective address is stored in General Purpose Register RA.
- If alignment checking is disabled, i.e., the alignment bit (AL) in the Machine Status Register (MSR) is 0, then the two low order bits of the effective address are ignored.
- If alignment checking is enabled, and the two low order bits are not b'00', then the hardware attempts to perform the unaligned storage access. If the hardware cannot perform the unaligned storage access, an Alignment Interrupt is generated.

The **lfsux** instruction has one syntax form and does not affect the Fixed Point Exception Register.

Parameters

FRT	Specifies target floating point register where data is stored.
RA	Specifies source general purpose register for EA calculation and possible address update.
RB	Specifies source general purpose register for EA calculation.

Examples

1. To load the single precision contents of storage into FPR 6 and store the effective address in GPR 5:

```
.csect data[rw]
storage: .float 0x1
# Assume GPR 4 contains the address of csect data[rw].
# Assume GPR 5 contains the displacement of storage
# relative to .csect data[rw].
.csect text[pr]
lfsux 6,5,4
# FPR 6 now contains 0x3FF0 0000 0000 0000.
# GPR 5 now contains the storage address.
```

Implementation Specifics

This instruction is part of Application Development Toolkit in AIX Base.

Related Information

Understanding Floating Point Load Instructions on page 1–13.

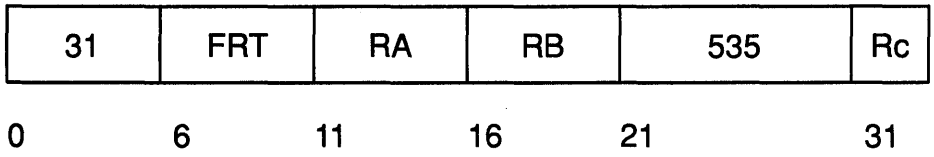
lfsx (Load Floating Point Single Indexed) Instruction

Purpose

Loads a floating point single precision number which is converted into a floating point double precision number into a floating point register.

Syntax

lfsx *FRT,RA,RB*



Description

The **lfsx** instruction converts a floating point single precision word in storage addressed by the effective address (EA) to floating point double precision and loads the result into Floating Point Register *FRT*.

The effective address (EA) is the sum of the contents of General Purpose Register *RA* and General Purpose Register *RB* if *RA* is not 0. If *RA* is 0, then the effective address (EA) is the contents of *RB*.

- If alignment checking is disabled, i.e., the alignment bit (AL) in the Machine Status Register (MSR) is 0, then the two low order bits of the effective address are ignored.
- If alignment checking is enabled, and the two low order bits are not b'00', then the hardware attempts to perform the unaligned storage access. If the hardware cannot perform the unaligned storage access, an Alignment Interrupt is generated.

The **lfsx** instruction has one syntax form and does not affect the Fixed Point Exception Register.

Parameters

FRT	Specifies target floating point register where data is stored.
RA	Specifies source general purpose register for EA calculation.
RB	Specifies source general purpose register for EA calculation.

Examples

1. To load the single precision contents of storage into FPR 6:

```
storage: .float 0x1.
# Assume GPR 4 contains the address of storage.
lfsx 6,0,4
# FPR 6 now contains 0x3FF0 0000 0000 0000.
```

Implementation Specifics

This instruction is part of Application Development Toolkit in AIX Base.

Related Information

Understanding Floating Point Load Instructions on page 1–13.

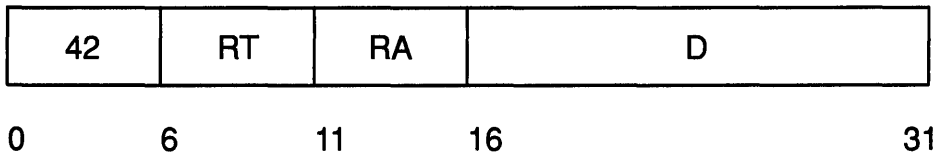
lha (Load Half Algebraic) Instruction

Purpose

Loads a halfword of data from a specified location in memory into a general purpose register and copies bit 0 of the halfword into the remaining 16 bits of the general purpose register.

Syntax

lha *RT,D(RA)*



Description

The **lha** instruction loads a halfword in storage from a specified location in memory addressed by the effective address (EA) into bits 16–31 of the target General Purpose Register *RT* and copies bit 0 of the halfword into bits 0–15 of General Purpose Register *RT*.

The effective address (EA) is the sum of the contents of General Purpose Register *RA* and *D*, a 16-bit signed two's complement integer sign extended to 32 bits, if *RA* is not 0. If *RA* is 0, then the effective address (EA) is *D*.

- If alignment checking is disabled, i.e., the alignment bit (AL) in the Machine Status Register (MSR) is 0, then the low order bit of the effective address is ignored.
- If alignment checking is enabled, and the low order bit is not b'0', then the hardware attempts to perform the unaligned storage access. If the hardware cannot perform the unaligned storage access, an Alignment Interrupt is generated.

The **lha** instruction has one syntax form and does not affect the Fixed Point Exception Register or Condition Register Field 0.

Parameters

RT	Specifies target general purpose register where result of operation is stored.
D	16-bit signed two's complement integer sign extended to 32 bits for EA calculation.
RA	Specifies source general purpose register for EA calculation.

Examples

1. To load a halfword of data into bits 16–31 of GPR 6 and copy bit 0 of the halfword into bits 0–15 of GPR 6:

```
.csect data[rw]
storage: .short 0xffff
# Assume GPR 5 contains the address of csect data[rw].
.csect text[pr]
lha 6,storage(5)
# GPR 6 now contains 0xffff ffff.
```

Implementation Specifics

This instruction is part of Application Development Toolkit in AIX Base.

Related Information

Understanding Fixed Point Load Instructions on page 1–6.

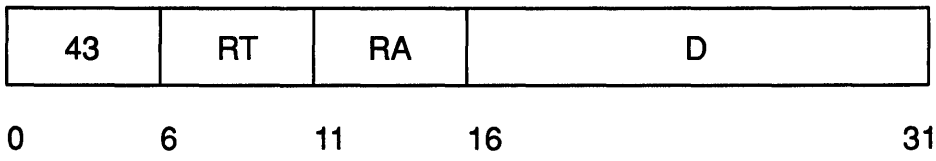
lhau (Load Half Algebraic With Update) Instruction

Purpose

Loads a halfword of data from a specified location in memory into a general purpose register, copies bit 0 of the halfword into the remaining 16 bits of the general purpose register, and possibly places the address in another general purpose register.

Syntax

lhau *RT,D(RA)*



Description

The **lhau** instruction loads a halfword in storage from a specified location in memory addressed by the effective address (EA) into bits 16–31 of the target General Purpose Register *RT* and copies bit 0 of the halfword into bits 0–15 of General Purpose Register *RT*.

The effective address (EA) is the sum of the contents of General Purpose Register *RA* and *D*, a 16-bit signed two's complement integer sign extended to 32 bits, if *RA* is not 0. If *RA* is 0, then the effective address (EA) is *D*.

- If *RA* does not equal *RT* and *RA* does not equal 0, and the storage acces does not cause an Alignment Interrupt or a Data Storage Interrupt, then the effective address (EA) is placed into General Purpose Register *RA*.
- If alignment checking is disabled, i.e., the alignment bit (AL) in the Machine Status Register (MSR) is 0, then the low order bit of the effective address is ignored.
- If alignment checking is enabled, and the low order bit is not b'0', then the hardware attempts to perform the unaligned storage access. If the hardware cannot perform the unaligned storage access, an Alignment Interrupt is generated.

The **lhau** instruction has one syntax form and does not affect the Fixed Point Exception Register or Condition Register Field 0.

Parameters

RT	Specifies target general purpose register where result of operation is stored.
D	16-bit signed two's complement integer sign extended to 32 bits for EA calculation.
RA	Specifies source general purpose register for EA calculation and possible address update.

Examples

1. To load a halfword of data into bits 16–31 of GPR 6, copy bit 0 of the halfword into bits 0–15 of GPR 6, and store the effective address in GPR 5:

```
.csect data[rw]
storage: .short 0xffff
# Assume GPR 5 contains the address of csect data[rw].
.csect text[pr]
lhau 6,storage(5)
# GPR 6 now contains 0xffff ffff.
# GPR 5 now contains the address of storage.
```

Implementation Specifics

This instruction is part of Application Development Toolkit in AIX Base.

Related Information

Understanding Fixed Point Load with Update Instructions on page 1–6.

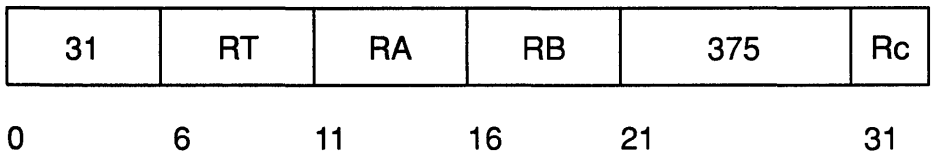
lhaux (Load Half Algebraic With Update Indexed) Instruction

Purpose

Loads a halfword of data from a specified location in memory into a general purpose register, copies bit 0 of the halfword into the remaining 16 bits of the general purpose register, and possibly places the address in another general purpose register.

Syntax

lhaux *RT,RA,RB*



Description

The **lhaux** instruction loads a halfword in storage from a specified location in memory addressed by the effective address (EA) into bits 16–31 of the target General Purpose Register *RT* and copies bit 0 of the halfword into bits 0–15 of General Purpose Register *RT*.

The effective address (EA) is the sum of the contents of General Purpose Register *RA* and General Purpose Register *RB* if *RA* is not 0. If *RA* is 0, then the effective address (EA) is the contents of *RB*.

- If *RA* does not equal *RT* and *RA* does not equal 0, and the storage access does not cause an Alignment Interrupt or a Data Storage Interrupt, then the effective address (EA) is placed into General Purpose Register *RA*.
- If alignment checking is disabled, i.e., the alignment bit (AL) in the Machine Status Register (MSR) is 0, then the low order bit of the effective address is ignored.
- If alignment checking is enabled, and the low order bit is not b'0', then the hardware attempts to perform the unaligned storage access. If the hardware cannot perform the unaligned storage access, an Alignment Interrupt is generated.

The **lhaux** instruction has one syntax form and does not affect the Fixed Point Exception Register.

Parameters

RT	Specifies target general purpose register where result of operation is stored.
RA	Specifies first source general purpose register for EA calculation and possible address update.
RB	Specifies second source general purpose register for EA calculation.

Examples

1. To load a halfword of data into bits 16–31 of GPR 6, copy bit 0 of the halfword into bits 0–15 of GPR 6, and store the effective address in GPR 5:

```
.csect data[rw]
storage: .short 0xffff
# Assume GPR 5 contains the address of csect data[rw].
# Assume GPR 4 contains the displacement of storage relative
# to data[rw].
.csect text[pr]
lhaux 6,5,4
# GPR 6 now contains 0xffff ffff.
# GPR 5 now contains the storage address.
```

Implementation Specifics

This instruction is part of Application Development Toolkit in AIX Base.

Related Information

Understanding Fixed Point Load with Update Instructions on page 1–6.

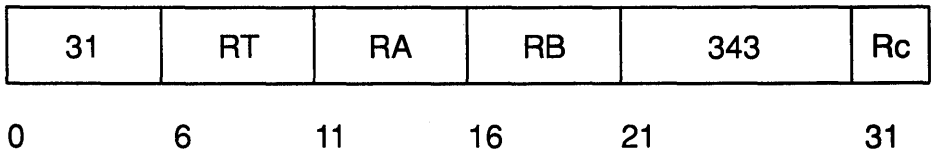
lhax (Load Half Algebraic Indexed) Instruction

Purpose

Loads a halfword of data from a specified location in memory into a general purpose register and copies bit 0 of the halfword into the remaining 16 bits of the general purpose register.

Syntax

lhax *RT,RA,RB*



Description

The **lhax** instruction loads a halfword in storage from a specified location in memory addressed by the effective address (EA) into bits 16–31 of the target General Purpose Register *RT* and copies bit 0 of the halfword into bits 0–15 of General Purpose Register *RT*.

The effective address (EA) is the sum of the contents of General Purpose Register *RA* and General Purpose Register *RB* if *RA* is not 0. If *RA* is 0, then the effective address (EA) is the contents of *RB*.

- If alignment checking is disabled (the alignment bit (AL) in the Machine Status Register (MSR) is 0) then the low order bit of the effective address is ignored.
- If alignment checking is enabled, and the low order bit is not b'0', then the hardware attempts to perform the unaligned storage access. If the hardware cannot perform the unaligned storage access, an Alignment Interrupt is generated.

The **lhax** instruction has one syntax form and does not affect the Fixed Point Exception Register.

Parameters

RT	Specifies target general purpose register where result of operation is stored.
RA	Specifies source general purpose register for EA calculation.
RB	Specifies source general purpose register for EA calculation.

Examples

1. To load a halfword of data into bits 16–31 of GPR 6 and copy bit 0 of the halfword into bits 0–15 of GPR 6:

```
.csect data[rw]
.short 0x1
# Assume GPR 5 contains the address of csect data[rw].
# Assume GPR 4 contains the displacement of the halfword
# relative to data[rw].
.csect text[pr]
lhax 6,5,4
# GPR 6 now contains 0x0000 0001.
```

Implementation Specifics

This instruction is part of Application Development Toolkit in AIX Base.

Related Information

Understanding Fixed Point Load Instructions on page 1–6.

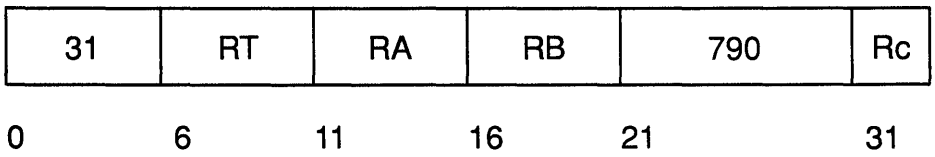
lhbrx (Load Half Byte Reverse Indexed) Instruction

Purpose

Loads a byte–reversed halfword of data from a specified location in memory into a general purpose register and sets the remaining 16 bits of the general purpose register to zero.

Syntax

lhbrx *RT,RA,RB*



Description

The **lhbrx** instruction loads bits 00–07 and bits 08–15 of the halfword in storage addressed by the effective address (EA) into bits 24–31 and bits 16–23 of General Purpose Register *RT* and sets bits 00–15 of General Purpose Register *RT* to 0.

The effective address (EA) is the sum of the contents of General Purpose Register *RA* and General Purpose Register *RB* if *RA* is not 0. If *RA* is 0, then the effective address (EA) is the contents of *RB*.

- If alignment checking is disabled, i.e., the alignment bit (AL) in the Machine Status Register (MSR) is 0, then the low order bit of the effective address is ignored.
- If alignment checking is enabled, and the low order bit is not b'0', then the hardware attempts to perform the unaligned storage access. If the hardware cannot perform the unaligned storage access, an Alignment Interrupt is generated.

The **lha** instruction has one syntax form and does not affect the Fixed Point Exception Register or Condition Refister Field 0.

Parameters

RT	Specifies target general purpose register where result of operation is stored.
RA	Specifies source general purpose register for EA calculation.
RB	Specifies source general purpose register for EA calculation.

Examples

1. To load bits 00–07 and bits 08–15 of the halfword in storage into bits 24–31 and bits 16–23 of GPR 6 and set bits 00–15 of GPR 6 to 0:

```
.csect data[rw]
.short 0x7654
# Assume GPR 4 contains the address of csect data[rw].
# Assume GPR 5 contains the displacement relative
# to data[rw].
.csect text[pr]
lhbrx 6,5,4
# GPR 6 now contains 0x0000 5476.
```

Implementation Specifics

This instruction is part of Application Development Toolkit in AIX Base.

Related Information

Understanding Fixed Point Load Instructions on page 1–6.

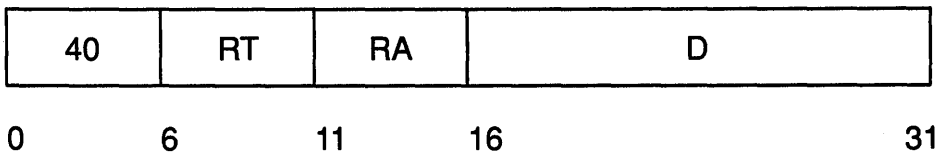
lhz (Load Half And Zero) Instruction

Purpose

Loads a halfword of data from a specified location in memory into a general purpose register and sets the remaining 16 bits to 0.

Syntax

lhz *RT,D(RA)*



Description

The **lhz** instruction loads a halfword in storage from a specified location in memory addressed by the effective address (EA) into bits 16–31 of the target General Purpose Register *RT* and sets bits 0–15 of General Purpose Register *RT* to zero.

The effective address (EA) is the sum of the contents of General Purpose Register *RA* and *D*, a 16-bit signed two's complement integer sign extended to 32 bits, if *RA* is not 0. If *RA* is 0, then the effective address (EA) is *D*.

- If alignment checking is disabled, i.e., the alignment bit (AL) in the Machine Status Register (MSR) is 0, then the low order bit of the effective address is ignored.
- If alignment checking is enabled, and the low order bit is not b'0', then the hardware attempts to perform the unaligned storage access. If the hardware cannot perform the unaligned storage access, an Alignment Interrupt is generated.

The **lhz** instruction has one syntax form and does not affect the Fixed Point Exception Register or Condition Register Field 0.

Parameters

RT	Specifies target general purpose register where result of operation is stored.
D	16-bit signed two's complement integer sign extended to 32 bits for EA calculation.
RA	Specifies source general purpose register for EA calculation.

Examples

1. To load a halfword of data into bits 16–31 of GPR 6 and set bits 0–15 of GPR 6 to 0:

```
.csect data[rw]
storage: .short 0xffff
# Assume GPR 4 holds the address of csect data[rw].
.csect text[pr]
lhz 6,storage(4)
# GPR 6 now holds 0x0000 ffff.
```

Implementation Specifics

This instruction is part of Application Development Toolkit in AIX Base.

Related Information

Understanding Fixed Point Load Instructions on page 1–6.

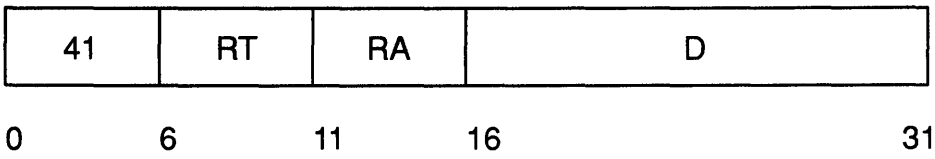
lhzu (Load Half And Zero With Update) Instruction

Purpose

Loads a halfword of data from a specified location in memory into a general purpose register, sets the remaining 16 bits of the general purpose register to zero, and possibly places the address in another general purpose register.

Syntax

lhzu *RT,D(RA)*



Description

The **lhzu** instruction loads a halfword in storage from a specified location in memory addressed by the effective address (EA) into bits 16–31 of the target General Purpose Register *RT* and sets bits 0–15 of General Purpose Register *RT* to zero.

The effective address (EA) is the sum of the contents of General Purpose Register *RA* and *D*, a 16-bit signed two's complement integer sign extended to 32 bits, if *RA* is not 0. If *RA* is 0, then the effective address (EA) is *D*.

- If *RA* does not equal *RT* and *RA* does not equal 0, and the storage access does not cause an Alignment Interrupt or a Data Storage Interrupt, then the effective address (EA) is placed into General Purpose Register *RA*.
- If alignment checking is disabled, i.e., the alignment bit (AL) in the Machine Status Register (MSR) is 0, then the low order bit of the effective address is ignored.
- If alignment checking is enabled, and the low order bit is not b'0', then the hardware attempts to perform the unaligned storage access. If the hardware cannot perform the unaligned storage access, an Alignment Interrupt is generated.

The **lhzu** instruction has one syntax form and does not affect the Fixed Point Exception Register or Condition Register Field 0.

Parameters

RT	Specifies target general purpose register where result of operation is stored.
D	16-bit signed two's complement integer sign extended to 32 bits for EA calculation.
RA	Specifies source general purpose register for EA calculation and possible address update.

Examples

1. To load a halfword of data into bits 16–31 of GPR 6, set bits 0–15 of GPR 6 to zero, and store the effective address in GPR 4:

```
.csect data[rw]
.short 0xffff
# Assume GPR 4 contains the address of csect data[rw].
.csect text[pr]
lhzu 6,0(4)
# GPR 6 now contains 0x0000 ffff.
```

Implementation Specifics

This instruction is part of Application Development Toolkit in AIX Base.

Related Information

Understanding Fixed Point Load with Update Instructions on page 1–6.

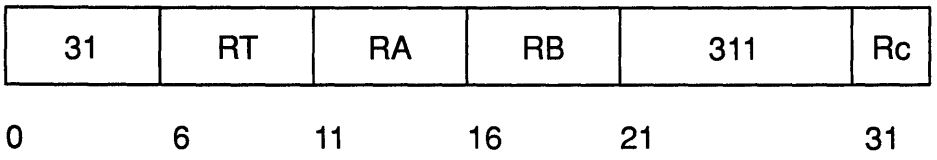
lhzux (Load Half And Zero With Update Indexed) Instruction

Purpose

Loads a halfword of data from a specified location in memory into a general purpose register, sets the remaining 16 bits of the general purpose register to zero, and possibly places the address in another general purpose register.

Syntax

lhzux *RT,RA,RB*



Description

The **lhzux** instruction loads a halfword in storage from a specified location in memory addressed by the effective address (EA) into bits 16–31 of the target General Purpose Register *RT* and sets bits 0–15 of General Purpose Register *RT* to zero.

The effective address (EA) is the sum of the contents of General Purpose Register *RA* and General Purpose Register *RB* if *RA* is not 0. If *RA* is 0, then the effective address (EA) is the contents of *RB*.

- If *RA* does not equal *RT* and *RA* does not equal 0, and the storage access does not cause an Alignment Interrupt or a Data Storage Interrupt, then the effective address (EA) is placed into General Purpose Register *RA*.
- If alignment checking is disabled, i.e., the alignment bit (AL) in the Machine Status Register (MSR) is 0, then the low order bit of the effective address is ignored.
- If alignment checking is enabled, and the low order bit is not b'0', then the hardware attempts to perform the unaligned storage access. If the hardware cannot perform the unaligned storage access, an Alignment Interrupt is generated.

The **lhzux** instruction has one syntax form and does not affect the Fixed Point Exception Register.

Parameters

RT	Specifies target general purpose register where result of operation is stored.
RA	Specifies source general purpose register for EA calculation and possible address update.
RB	Specifies source general purpose register for EA calculation.

Examples

1. To load a halfword of data into bits 16–31 of GPR 6, set bits 0–15 of GPR 6 to zero, and store the effective address in GPR 5:

```
.csect data[rw]
storage: .short 0xffff
# Assume GPR 5 contains the address of csect data[rw].
# Assume GPR 4 contains the displacement of storage
# relative to data[rw].
.csect text[pr]
lhzux 6,5,4
# GPR 6 now contains 0x0000 ffff.
# GPR 5 now contains the storage address.
```

Implementation Specifics

This instruction is part of Application Development Toolkit in AIX Base.

Related Information

Understanding Fixed Point Load with Update Instructions on page 1–6.

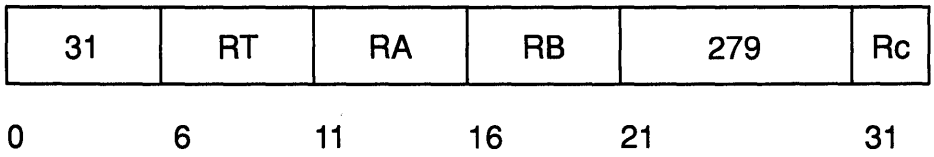
lhzx (Load Half And Zero Indexed) Instruction

Purpose

Loads a halfword of data from a specified location in memory into a general purpose register and sets the remaining 16 bits of the general purpose register to zero.

Syntax

lhzx *RT,RA,RB*



Description

The **lhzx** instruction loads a halfword in storage from a specified location in memory addressed by the effective address (EA) into bits 16–31 of the target General Purpose Register *RT* and sets bits 0–15 of General Purpose Register *RT* to zero.

The effective address (EA) is the sum of the contents of General Purpose Register *RA* and General Purpose Register *RB* if *RA* is not 0. If *RA* is 0, then the effective address (EA) is the contents of *RB*.

- If alignment checking is disabled, i.e., the alignment bit (AL) in the Machine Status Register (MSR) is 0, then the low order bit of the effective address is ignored.
- If alignment checking is enabled, and the low order bit is not b'0', then the hardware attempts to perform the unaligned storage access. If the hardware cannot perform the unaligned storage access, an Alignment Interrupt is generated.

The **lhzx** instruction has one syntax form and does not affect the Fixed Point Exception Register or Condition Register Field 0.

Parameters

RT	Specifies target general purpose register where result of operation is stored.
RA	Specifies source general purpose register for EA calculation.
RB	Specifies source general purpose register for EA calculation.

Examples

1. To load a halfword of data into bits 16–31 of GPR 6 and set bits 0–15 of GPR 6 to zero:

```
.csect data[rw]
.short 0xffff
.csect text[pr]
# Assume GPR 5 contains the address of csect data[rw].
# Assume 0xffff is the halfword located at displacement 0.
# Assume GPR 4 contains 0x0000 0000.
lhzx 6,5,4
# GPR 6 now contains 0x0000 ffff.
```

Implementation Specifics

This instruction is part of Application Development Toolkit in AIX Base.

Related Information

Understanding Fixed Point Load Instructions on page 1–6.

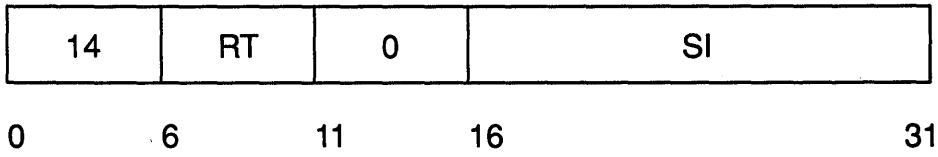
lil (Load Immediate Lower) Instruction

Purpose

Loads a 16–bit signed integer into a general purpose register.

Syntax

lil *RT,SI*



Descritpion

The **lil** instruction loads the 16–bit two's complement (sign extended to 32 bits) signed integer *SI* into the target General Purpose Register *RT*. This instruction has the same effect as the **cal** instruction used with the General Purpose Register *RA* parameter equal to zero.

The **lil** instruction has one syntax form and does not affect Condition Register Field 0 or the Fixed Point Exception Register.

Parameters

RT	Specifies target general purpose register for operation.
SI	Specifies 16–bit signed immediate integer for operation.

Examples

1. To load 0xFFFF FFFF into GPR 6:

```
lil 6,0xFFFFFFFF
# GPR 6 now contains 0xFFFF FFFF
# This instruction has the same effect as
#     cal 6,0xFFFFFFFF(0).
```

Implementation Specifics

This instruction is part of Application Development Toolkit in AIX Base.

Related Information

The **cal** (Compute Address Lower) Instruction.

Understanding Fixed Point Load Instructions on page 1–6.

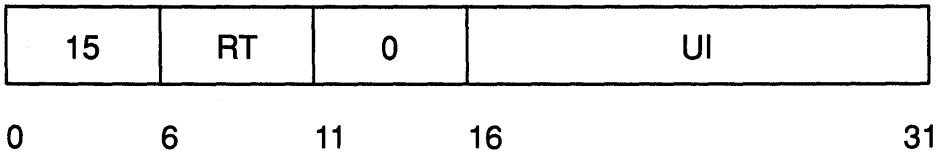
liu (Load Immediate Upper) Instruction

Purpose

Loads a 16–bit unsigned integer into the upper half of a general purpose register.

Syntax

liu *RT,UI*



Descritpion

The **liu** instruction loads the concatenation of the 16–bit unsigned integer *UI* and 'x0000' into the target General Purpose Register *RT*. This instruction has the same effect as the **cau** instruction used with the General Purpose Register *RA* parameter equal to zero.

The **liu** instruction has one syntax form and does not affect Condition Register Field 0 or the Fixed Point Exception Register.

Parameters

RT	Specifies target general purpose register for operation.
UI	Specifies 16–bit unsigned immediate integer for operation.

Examples

1. To load 0xFFFF 0000 into GPR 6:

```
liu 6, 0xFFFF
# GPR 6 now contains 0xFFFF 0000.
# This instruction has the same effect as
#     cau 6,0,0xFFFF.
```

Implementation Specifics

This instruction is part of Application Development Toolkit in AIX Base.

Related Information

The **cau** (Compute Address Upper) Instruction.

Understanding Fixed Point Load Instructions on page 1–6.

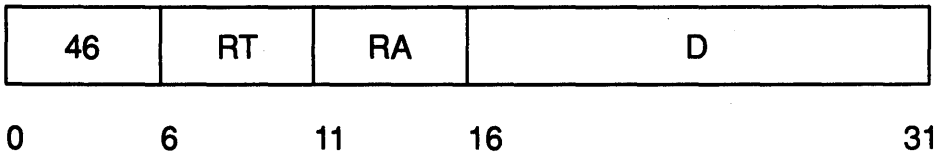
lm (Load Multiple) Instruction

Purpose

Loads consecutive words at a specified location into more than one general purpose register.

Syntax

lm *RT,D(RA)*



Description

The **lm** instruction loads *N* consecutive words starting at the calculated effective address (EA) into a number of General Purpose Registers, starting at General Purpose Register *RT* and filling all GPRs through General Purpose Register 31. *N* is equal to 32–*RT* field, the total number of consecutive words that will be placed in consecutive registers.

The effective address (EA) is the sum of the contents of General Purpose Register *RA* and *D* if *RA* is not 0. If *RA* is 0, then EA is *D*.

- If *RA* is not equal to 0, it is not written into if it is in the range to be loaded. The data that would have normally been written into it is discarded. The operation continues normally.
- If alignment checking is disabled, i.e., the alignment bit (AL) in the Machine Status Register (MSR) is 0, then the low order two bits of the EA are ignored.
- If Alignment checking is enabled (MSR(AL) = 1), and the low order two bits are not b'00', then an Alignment Interrupt is generated.

The **lm** instruction has one syntax and does not affect the Fixed Point Exception Register or Condition Register Field 0.

Note: This instruction is interruptible due to a data storage interrupt. When such an interrupt occurs, the instruction should be restarted from the beginning.

Parameters

RT	Specifies starting target general purpose register for operation.
D	Specifies a 16-bit signed two's complement integer sign extended to 32 bits for EA calculation
RA	Specifies source general purpose register for EA calculation.

Examples

1. To load data into GPR 29 and GPR 31:

```
.csect data[rw]
.long 0x8971
.long -1
.long 0x7ffe c100
# Assume GPR 30 contains the address of csect data[rw].
.csect text[pr]
lm 29,0(30)
# GPR 29 now contains 0x0000 8971.
# GPR 30 now contains the address of csect data[rw].
# GPR 31 now contains 0x7ffe c100.
```

Implementation Specifics

This instruction is part of Application Development Toolkit in AIX Base.

Related Information

Understanding Fixed Point Load Instructions on page 1–6.

lscbx (Load String And Compare Byte Indexed) Instruction

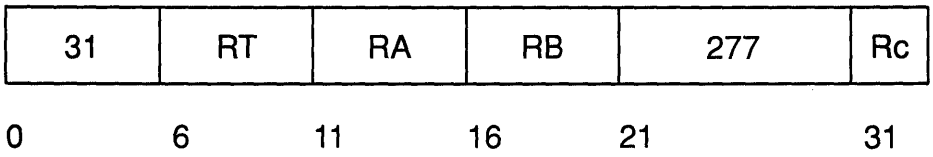
Purpose

Loads consecutive bytes in storage into consecutive registers.

Syntax

lscbx *RT,RA,RB*

lscbx. *RT,RA,RB*



Description

The **lscbx** instruction loads *N* consecutive bytes addressed by EA into General Purpose Register *RT*, starting with the leftmost byte in register *RT*, through *RT* + *NR* – 1, wrapping around back through GPR 0 if required, until either a byte match is found with XER16–23 or *N* bytes have been loaded. If a byte match is found, then that byte is also loaded.

The effective address (EA) is the sum of the contents of General Purpose Register *RA* and the address stored in General Purpose Register *RB* if *RA* is not 0. If *RA* is 0, then EA is the contents of *RB*.

- XER(16–23) contains the byte to be compared.
- XER(25–31) contains the byte count before the instruction is invoked and the number of bytes loaded after the instruction has completed.
- If XER(25–31) = 0, General Purpose Register *RT* is not altered.
- *N* is XER(25–31), which is the number of bytes to load.
- *NR* is ceil(*N*/4), which is the total number of registers required to contain the consecutive bytes.

Bytes are always loaded left to right in the register. In the case when a match was found before *N* bytes were loaded, the contents of the rightmost bytes not loaded of that register and the contents of all succeeding registers up to and including register *RT* + *NR* – 1 are undefined. Also, no reference is made to storage after the matched byte is found. In the case when a match was not found, the contents of the rightmost byte(s) not loaded of register *RT* + *NR* – 1 is undefined.

General Purpose Registers *RA* (if *RA* is not 0) and *RB*, if in the range to be loaded, are not written into. The data that would have been written into them is discarded, and the operation continues normally. If the byte in XER(16–23) compares with any of the four bytes that would have been loaded into General Purpose Register *RA* or *RB*, but are being discarded for restartability, the EQ bit in the Condition Register and the count returned in XER(25–31) are undefined. The MQ Register is not affected by this operation.

The **lscbx** instruction has two syntax forms. Each syntax form has a different effect on Condition Register Field 0.

Syntax form	Overflow Exception (OE)	Fixed Point Exception Register	Record bit (Rc)	Condition Register Field 0
lscbx	None	XER(25–31) = # of bytes loaded	0	None
lscbx.	None	XER(25–31) = # of bytes loaded	1	LT,GT,EQ,SO

The two syntax forms of the **lscbx** instruction places the number of bytes loaded into Fixed Point Exception Register (XER) bits 25–31. If the syntax form sets the Record (Rc) bit to 1, the instruction affects the Less Than (LT) zero, Greater Than (GT) zero, Equal To (EQ) zero, and Summary Overflow (SO) bits in Condition Register Field 0. If Rc = 1 and XER(25–31) = 0, then Condition Register Field 0 is undefined. If Rc = 1 and XER(25–31) <> 0, then CR field 0 is set as follows:

LT, GT, EQ, SO = b'00'||match||XER(SO)

Note: This instruction is interruptible due to a data storage interrupt. When such an interrupt occurs, the instruction is restarted from the beginning.

Parameters

RT Specifies the starting target general purpose register.

RA Specifies source general purpose register for EA calculation.

RB Specifies source general purpose register for EA calculation.

Examples

1. To load consecutive bytes into GPRs 6, 7, and 8:

```
.csect data[rw]
string: "Hello, world"
# Assume XER16–23 = 'a.
# Assume XER25–31 = 9.
# Assume GPR 5 contains the address of csect data[rw].
# Assume GPR 4 contains the displacement of string relative
# to csect data[rw].
.csect text[pr]
lscbx 6,5,4
# GPR 6 now contains 0x4865 6c6c.
# GPR 7 now contains 0x6f2c 2077.
# GPR 8 now contains 0x6fXX XXXX.
```

2. To load consecutive bytes into GPRs 6, 7, and 8:

```
# Assume XER16–23 = 'e.
# Assume XER25–31 = 9.
# Assume GPR 5 contains the address of csect data[rw].
# Assume GPR 4 contains the displacement of string relative
# to csect data[rw].
.csect text[pr]
lscbx. 6,5,4
# GPR 6 now contains 0x4865 XXXX.
# GPR 7 now contains 0xXXXX XXXX.
# GPR 8 now contains 0xXXXX XXXX.
# XER25–31 = 2.
# CRF 0 now contains 0x2.
```

Implementation Specifics

This instruction is part of Application Development Toolkit in AIX Base.

Related Information

Understanding Fixed Point String Instructions on page 1–8.

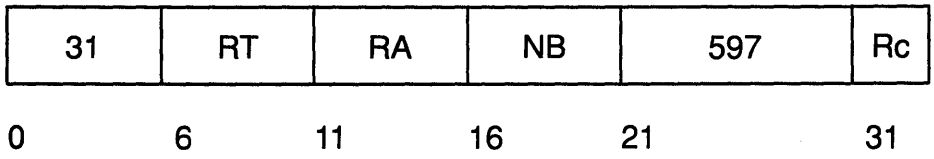
lsi (Load String Immediate) Instruction

Purpose

Loads consecutive bytes in storage from a specified location in memory into consecutive general purpose registers.

Syntax

lsi *RT,RA,NB*



Description

The **lsi** instruction loads *N* consecutive bytes in storage addressed by the effective address (EA) into General Purpose Register *RT,* starting with the leftmost byte, through General Purpose Register *RT+NR–*1, wrapping around back through General Purpose Register 0 if required.

The effective address (EA) is the contents of General Purpose Register *RA* if *RA* is not 0. If *RA* is 0, then the effective address (EA) is 0.

- *NB* is the byte count.
- *RT* is the starting General Purpose Register.
- *N* is *NB*, which is the number of bytes to load. If *NB* is 0, then *N* is 32.
- NR is ceiling(N/4), which is the number of General Purpose Registers to receive data.
- If General Purpose Register *RT* + *NR* – 1 is only partially filled on the left, the rightmost byte(s) of that General Purpose Register are set to zero.
- If *RA* is in the range to be loaded, and if *RA* is not equal to 0, then General Purpose Register *RA* is not written into by this instruction. The data that would have been written into it is discarded, and the operation continues normally.
- The contents of the MQ Register are not effected by this operation.

The **lsi** instruction has one syntax form and does not affect the Fixed Point Exception General Purpose Register or Condition Register Field 0.

Note: This instruction is interruptible due to a data storage interrupt. When such an interrupt occurs, the instruction is restarted from the beginning.

Parameters

RT	Specifies starting General Purpose Register of stored data.
RA	Specifies General Purpose Register for EA calculation.
NB	Specifies byte count.

Examples

1. To load the bytes contained in a location in memory addressed by GPR 7 into GPR 6:

```
.csect data[rw]
.string "Hello, World"
# Assume GPR 7 contains the address of csect data[rw].
.csect text[pr]
lsi 6,7,0x6
# GPR 6 now contains 0x4865 6c6c.
```

Implementation Specifics

This instruction is part of Application Development Toolkit in AIX Base.

Related Information

Understanding Fixed Point String Instructions on page 1–8.

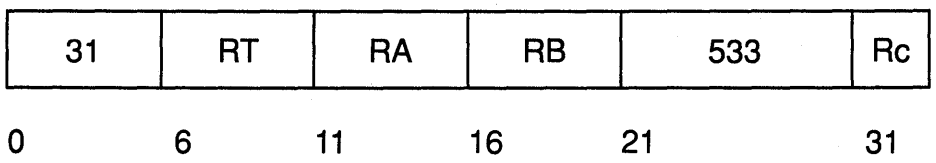
lsx (Load String Indexed) Instruction

Purpose

Loads consecutive bytes in storage from a specified location in memory into consecutive general purpose registers.

Syntax

lsx *RT,RA,RB*



Description

The **lsx** instruction loads *N* consecutive bytes in storage addressed by the effective address (EA) into General Purpose Register *RT,* starting with the leftmost byte, through General Purpose Register *RT* + *NR* – 1, wrapping around back through General Purpose Register 0 if required.

The effective address (EA) is the sum of the contents of General Purpose Register *RA* and the address stored in General Purpose Register *RB* if RA is not 0. If *RA* is 0, then effective address (EA) is the contents of *RB*.

- XER(25–31) contain the byte count.
- *RT* is the starting General Purpose Register.
- *N* is XER(25–31), which is the number of bytes to load.
- *NR* is ceiling(N/4), which is the number of registers to receive data.
- The contents of the MQ Register are not effected by this operation.
- If XER(25–31) = 0, General Purpose Register *RT* is not altered.
- If General Purpose Register *RT* + *NR* – 1 is only partially filled on the left, the rightmost byte(s) of that General Purpose Register are set to zero.
- If they are in the range to be loaded, and if *RA* is not equal to 0, then General Purpose Registers *RA* and *RB* is not written into by this instruction. The data that would have been written into them is discarded, and the operation continues normally.

The **lsx** instruction has one syntax form and does not affect the Fixed Point Exception Register or Condition Register Field 0.

Note: This instruction is interruptible due to a data storage interrupt. When such an interrupt occurs, the instruction is restarted from the beginning.

Parameters

RT	Specifies starting General Purpose Register of stored data.
RA	Specifies General Purpose Register for EA calculation.
RB	Specifies General Purpose Register for EA calculation.

Examples

1. To load the bytes contained in a location in memory addressed by GPR 5 into GPR 6:

```
# Assume XER25–31 = 4.
csect data[rw]
storage: .string "Hello, world"
# Assume GPR 4 contains the displacement of storage
# relative to data[rw].
# Assume GPR 5 contains the address of csect data[rw].
.csect text[pr]
lsx 6,5,4
# GPR 6 now contains 0x4865 6c6c.
```

Implementation Specifics

This instruction is part of Application Development Toolkit in AIX Base.

Related Information

Understanding Fixed Point String Instructions on page 1–8.

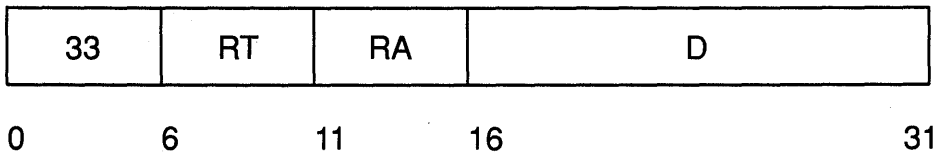
lu (Load With Update) Instruction

Purpose

Loads a word of data from a specified location in memory into a general purpose register and possibly places the effective address in a second general purpose register.

Syntax

lu *RT,D(RA)*



Description

The **lu** instruction loads a word in storage from a specified location in memory addressed by the effective address (EA) into the target General Purpose Register *RT*.

The effective address (EA) is the sum of the contents of General Purpose Register *RA* and *D*, a 16-bit signed two's complement integer sign extended to 32 bits, if *RA* is not 0. If *RA* is 0, then the effective address (EA) is *D*.

- If *RA* does not equal RT and *RA* does not equal 0, and the storage access does not cause an Alignment Interrupt or a Data Storage Interrupt, then the effective address (EA) is placed into General Purpose Register *RA*
- If alignment checking is disabled, i.e., the alignment bit (AL) in the Machine Status Register (MSR) is 0, then the two low order bits of the effective address are ignored.
- If alignment checking is enabled, and the two low order bits are not b'00', then the hardware attempts to perform the unaligned storage access. If the hardware cannot perform the unaligned storage access, an Alignment Interrupt is generated.

The **lu** instruction has one syntax form and does not affect the Fixed Point Exception Register or Condition Register Field 0.

Parameters

RT	Specifies target general purpose register where result of operation is stored.
D	16-bit signed two's complement integer sign extended to 32 bits for EA calculation.
RA	Specifies source general purpose register for EA calculation and possible address update.

Examples

1. To load a word from memory into GPR 6 and place the effective address in GPR 4:

```
.csect data[rw]
storage: .long 0xffdd 75ce
.csect text[pr]
# Assume GPR 4 contains address of csect data[rw].
lu 6,storage(4)
# GPR 6 now contains 0xffdd 75ce.
# GPR 4 now contains the storage address.
```

Implementation Specifics

This instruction is part of Application Development Toolkit in AIX Base.

Related Information

Understanding Fixed Point Load with Update Instructions on page 1–6.

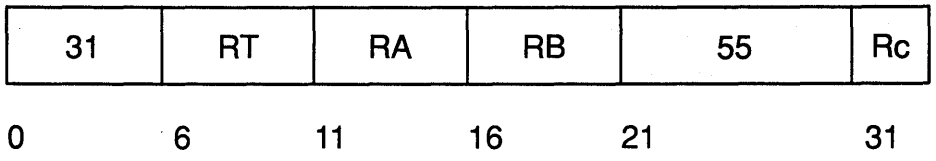
lux (Load With Update Indexed) Instruction

Purpose

Loads a word of data from a specified location in memory into a general purpose register and possibly places the effective address in a second general purpose register.

Syntax

lux *RT,RA,RB*



Description

The **lux** instruction loads a word in storage from a specified location in memory addressed by the effective address (EA) into the target General Purpose Register *RT*.

The effective address (EA) is the sum of the contents of General Purpose Register *RA* and General Purpose Register *RB* if *RA* is not 0. If *RA* is 0, then the effective address (EA) is the contents of *RB*.

- If *RA* does not equal RT and *RA* does not equal 0, and the storage access does not cause an Alignment Interrupt or a Data Storage Interrupt, then the effective address (EA) is placed into General Purpose Register *RA*.
- If alignment checking is disabled, i.e., the alignment bit (AL) in the Machine Status Register (MSR) is 0, then the two low order bits of the effective address are ignored.
- If alignment checking is enabled, and the two low order bits are not b'00', then the hardware attempts to perform the unaligned storage access. If the hardware cannot perform the unaligned storage access, an Alignment Interrupt is generated.

The **lux** instruction has one syntax form and does not affect the Fixed Point Exception Register or Condition Register Field 0.

Parameters

RT	Specifies target general purpose register where result of operation is stored.
RA	Specifies source general purpose register for EA calculation and possible address update.
RB	Specifies source general purpose register for EA calculation.

Examples

1. To load a word from memory into GPR 6 and place the effective address in GPR 5:

```
.csect data[rw]
storage: .long 0xffdd 75ce
# Assume GPR 5 contains the address of csect data[rw].
# Assume GPR 4 contains the displacement of storage
# relative to csect data[rw].
.csect text[pr]
lux 6,5,4
# GPR 6 now contains 0xffdd 75ce.
# GPR 5 now contains the storage address.
```

Implementation Specifics

This instruction is part of Application Development Toolkit in AIX Base.

Related Information

Understanding Fixed Point Load with Update Instructions on page 1–6.

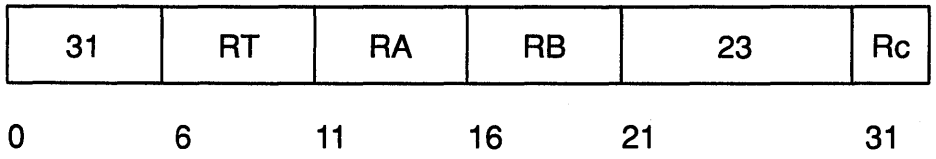
lx (Load Indexed) Instruction

Purpose

Loads a word of data from a specified location in memory into a general purpose register.

Syntax

lx *RT,RA,RB*



Description

The **lx** instruction loads a word in storage from a specified location in memory addressed by the effective address (EA) into the target General Purpose Register *RT*.

The effective address (EA) is the sum of the contents of General Purpose Register *RA* and General Purpose Register *RB* if *RA* is not 0. If *RA* is 0, then the effective address (EA) is the contents of *RB*.

- If alignment checking is disabled, i.e., the alignment bit (AL) in the Machine Status Register (MSR) is 0, then the two low order bits of the effective address are ignored.
- If alignment checking is enabled, and the two low order bits are not b'00', then the hardware attempts to perform the unaligned storage access. If the hardware cannot perform the unaligned storage access, an Alignment Interrupt is generated.

The **lx** instruction has one syntax form and does not affect the Fixed Point Exception Register or Condition Register Field 0.

Parameters

RT	Specifies target general purpose register where result of operation is stored.
RA	Specifies source general purpose register for EA calculation.
RB	Specifies source general purpose register for EA calculation.

Examples

1. To load a word from memory into GPR 6:

```
.csect data[rw]
.long 0xffdd 75ce
# Assume GPR 4 contains the displacement relative to
# csect data[rw].
# Assume GPR 5 contains the address of csect data[rw].
.csect text[pr]
lx 6,5,4
# GPR 6 now contains 0xffdd 75ce.
```

Implementation Specifics

This instruction is part of Application Development Toolkit in AIX Base.

Related Information

Understanding Fixed Point Load Instructions on page 1–6.

maskg (Mask Generate) Instruction

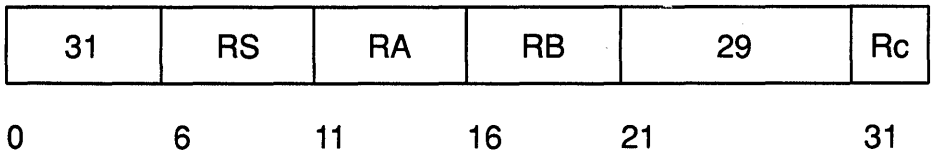
Purpose

Generates a mask of ones and zeros and loads it into a general purpose register.

Syntax

maskg *RA,RS,RB*

maskg. *RA,RS,RB*



Description

The **maskg** instruction generates a mask from a starting point defined by bits 27–31 of General Purpose Register *RS* to an end point defined by bits 27–31 of General Purpose Register *RB* and stores the mask in General Purpose Register *RA*.

- If the starting point bit is less than the end point bit + 1, then the bits between and including the starting point and the end point are set to ones. All other bits are set to zeros.
- If the starting point bit is the same as the end point bit + 1, then all 32 bits are set to ones.
- If the starting point bit is greater than the end point bit + 1, then all of the bits between and including the end point bit +1 and the starting point bit –1 are set to zeros. All other bits are set to ones.

The **maskg** instruction has two syntax forms. Each syntax form has a different effect on Condition Register Field 0.

Syntax form	Overflow Exception (OE)	Fixed Point Exception Register	Record bit (Rc)	Condition Register Field 0
maskg	None	None	0	None
maskg.	None	None	1	LT,GT,EQ,SO

The two syntax forms of the **maskg** instruction never affect the Fixed Point Exception Register. If the syntax form sets the Record (Rc) bit to 1, the instruction affects the Less Than (LT) zero, Greater Than (GT) zero, Equal To (EQ) zero, and Summary Overflow (SO) bits in Condition Register Field 0.

Parameters

RA	Specifies target general purpose register where result of operation is stored.
RS	Specifies source general purpose register for start of mask.
RB	Specifies source general purpose register for end of mask.

Examples

1. To generate a mask of 5 ones and store the result in GPR 6:

```
# Assume GPR 4 contains 0x0000 0014.
# Assume GPR 5 contains 0x0000 0010.
maskg 6,5,4
# GPR 6 now contains 0x0000 F800.
```

2. To generate a mask of 6 zeros with the remaining bits set to one, store the result in General Purpose Register 6, and set Condition Register Field 0 to reflect the result of the operation:

```
# Assume GPR 4 contains 0x0000 0010.
# Assume GPR 5 contains 0x0000 0017.
# Assume CR = 0.
maskg. 6,5,4
# GPR 6 now contains 0xFFFF 81FF.
# CR now contains 0x8000 0000.
```

Implementation Specifics

This instruction is part of Application Development Toolkit in AIX Base.

Related Information

Understanding Fixed Point Rotate Instructions on page 1–9.

maskir (Mask Insert From Register) Instruction

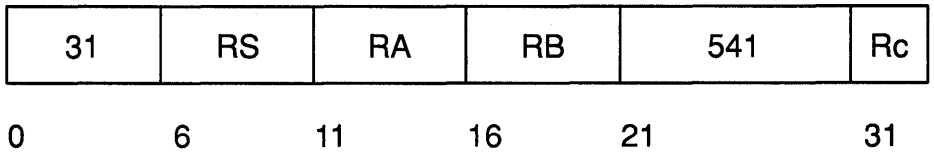
Purpose

Inserts the contents of one general purpose register into another general purpose register under control of a bit mask.

Syntax

maskir *RA,RS,RB*

maskir. *RA,RS,RB*



Description

The **maskir** stores the contents of General Purpose Register *RS* in General Purpose Register *RA* under control of the bit mask in General Purpose Register *RB*.

The **maskir** instruction has two syntax forms. Each syntax form has a different effect on Condition Register Field 0.

Syntax form	Overflow Exception (OE)	Fixed Point Exception Register	Record bit (Rc)	Condition Register Field 0
maskir	None	None	0	None
maskir.	None	None	1	LT,GT,EQ,SO

The two syntax forms of the **maskir** instruction never affect the Fixed Point Exception Register. If the syntax form sets the Record (Rc) bit to 1, the instruction affects the Less Than (LT) zero, Greater Than (GT) zero, Equal To (EQ) zero, and Summary Overflow (SO) bits in Condition Register Field 0.

Parameters

RA Specifies target general purpose register where result of operation is stored.

RS Specifies source general purpose register for operation.

RB Specifies source general purpose register for bit mask.

Examples

1. To insert the contents of GPR 5 into GPR 6 under control of the bit mask in GPR 4:

```
# Assume GPR 4 contains 0x8000 0000.
# Assume GPR 5 contains 0x7654 EF13.
maskir 6,5,4
# GPR 6 now contains 0x0000 0000.
```

2. To insert the contents of GPR 5 into GPR 6 under control of the bit mask in GPR 4 and set Condition Register Field 0 to reflect the result of the operation:

```
# Assume GPR 4 contains 0x7FFF FFFF.
# Assume GPR 5 contains 0xB004 3000.
# Assume CR = 0.
maskir. 6,5,4
# GPR 6 now contains 0x3004 3000.
# CR now contains 0x4000 0000.
```

Implementation Specifics

This instruction is part of Application Development Toolkit in AIX Base.

Related Information

Understanding Fixed Point Rotate Instructions on page 1–9.

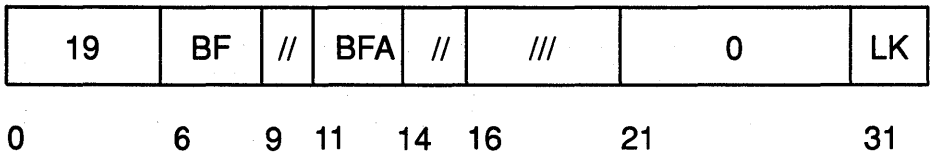
mcrf (Move Condition Register Field) Instruction

Purpose

Copies the contents of one Condition Register Field into another.

Syntax

mcrf *BF,BFA*



Description

The **mcrf** instruction copies the contents of the Condition Register Field specified by *BFA* into the Condition Register Field specified by *BF*. All other fields remain unaffected.

The **mcrf** instruction has one syntax form and does not affect Condition Register Field 0 or the Fixed Point Exception Register.

Parameters

BF	Specifies target Condition Register Field for operation.
BFA	Specifies source Condition Register Field for operation.

Examples

1. To copy the contents of Condition Register Field 3 into Condition Register Field 2:

```
# Assume Condition Register Field 3 holds b'0110'.
mcrf 2,3
# Condition Register Field 2 now holds b'0110'.
```

Implementation Specifics

This instruction is part of Application Development Toolkit in AIX Base.

Related Information

See the following article in *POWERstation and POWERserver Hardware Technical Reference — General Information*: Condition Register.

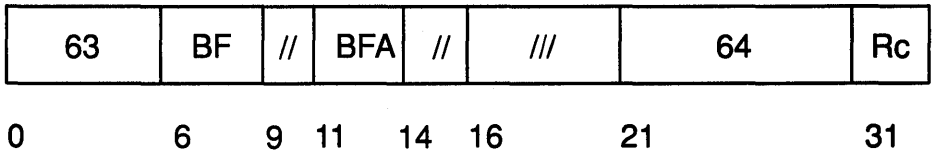
mcrfs (Move To Condition Register From FPSCR) Instruction

Purpose

Copies the bits from one field of the Floating Point Status and Control Register into the Condition Register.

Syntax

mcrfs *BF,BFA*



Description

The **mcrfs** instruction copies four bits of the Floating Point Status and Control Register (FPSCR) specified by *BFA* into Condition Register field *BF*. All other Condition Register bits are unchanged.

If the field specified by *BFA* contains reserved or undefined bits, then bits of zero value are supplied for the copy.

The **mcrfs** instruction has one syntax form and can set the bits of the Floating Point Status and Control Register.

BFA	FPSCR bits set
0	FX,OX
1	UX, ZX, XX, VXSNAN
2	VXISI, VXIDI, VXZDZ, VXIMZ
3	VXVC

Parameters

BF Specifies target Condition Register field where result of operation is stored.

BFA Specifies one of the FPSCR fields (0–7).

Examples

1. To copy bits from Floating Point Status and Control Register Field 4 into Condition Register Field 3:

```
# Assume FPSCR 4 contains b'0111'.
mcrfs 3,4
# Condition Register Field 3 contains b'0111'.
```

Implementation Specifics

This instruction is part of Application Development Toolkit in AIX Base.

Related Information

See the following article in *POWERstation and POWERserver Hardware Technical Reference — General Information*: Condition Register.

Understanding The Floating Point Status and Control Register on page 1–12.

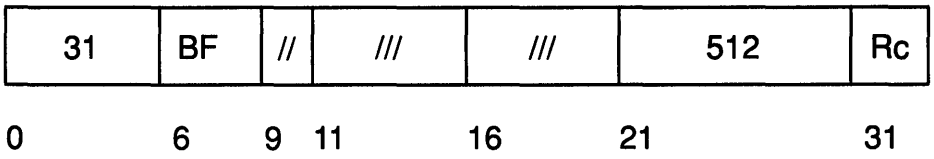
mcrxr (Move To Condition Register From XER) Instruction

Purpose

Copies the Summary Overflow bit, Overflow bit, Carry bit, and bit 3 from the Fixed Point Exception Register into a specified field of the Condtion Register.

Syntax

mcrxr *BF*



Description

The **mcrxr** copies the contents of Fixed Point Exception Register Field 0 bits 0–3 into Condition Register Field *BF* and resets Fixed Point Exception Register Field 0 to zeros.

The **mcrxr** instruction has one syntax form and resets Fixed Point Exception Register bits 0–3 to zero.

Parameters

BF Specifies target Condition Register field where result of operation is stored.

Examples

1. To copy the Summary Overflow bit, Overflow bit, Carry bit, and bit 3 from the Fixed Point Exception Register into field 4 of the Condition Register.

```
# Assume bits 0–3 of the Fixed Point Exception
# Register are set to b'1110'.
mcrxr 4
# Condition Register Field 4 now holds b'1110'.
```

Implementation Specifics

This instruction is part of Application Development Toolkit in AIX Base.

Related Information

See the following article in *POWERstation and POWERserver Hardware Technical Reference — General Information*: Condition Register.

Understanding Fixed Point Move To/From System Registers Instructions on page 1–11.

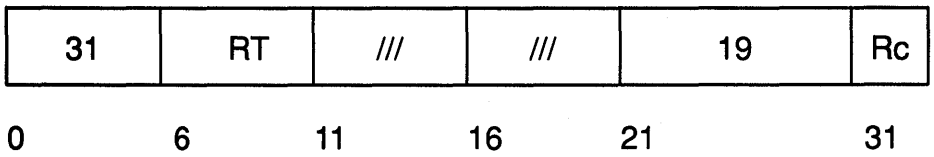
mfcr (Move From Condition Register) Instruction

Purpose

Copies the contents of the Condition Register into a general purpose register.

Syntax

mfcr *RT*



Description

The **mfcr** instruction copies the contents of the Condition Register into the target General Purpose Register *RT*.

The **mfcr** instruction has one syntax form and does not affect the Fixed Point Exception Register.

Parameters

RT Specifies target general purpose register where result of operation is stored.

Examples

1. To copy the Condition Register into GPR 6:

```
# Assume the Condition Register contains 0x4055 F605.
mfcr 6
# GPR 6 now contains 0x4055 F605.
```

Implementation Specifics

This instruction is part of Application Development Toolkit in AIX Base.

Related Information

See the following article in *POWERstation and POWERserver Hardware Technical Reference — General Information*: Condition Register.

Understanding Fixed Point Move To/From System Registers Instructions on page 1–11.

mffs (Move From FPSCR) Instruction

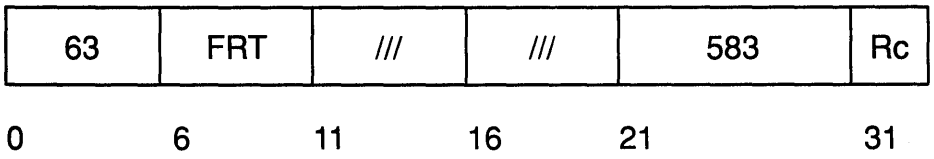
Purpose

Loads the contents of the Floating Point Status and Control Register into a Floating Point Register and fills the upper 32 bits with ones.

Syntax

mffs *FRT*

mffs. *FRT*



Description

The **mffs** instruction places the contents of the Floating Point Status and Control Register into bits 32–63 of Floating Point Register *FRT* and places 0xFFFF FFFF into bits 0–31 of Floating Point Register *FRT*.

The **mffs** instruction has two syntax forms. Each syntax form has a different effect on Condition Register Field 1.

Syntax form	FPSCR bits	Record bit (Rc)	Condition Register Field 1
mffs	None	0	None
mffs.	None	1	FX, FEX, VX, OX

The two syntax forms of the **mffs** instruction never affect the Floating Point Status and Control Register fields. If the syntax form sets the Record (Rc) bit to 1, the instruction affects the Floating Point Exception (FX), Floating Point Enabled Exception (FEX), Floating Invalid Operation Exception (VX), and Floating Point Overflow Exception (OX) bits in Condition Register Field 1.

Note: This instruction loads the contents of the Floating Point Status and Control Register into a Floating Point Register, loading ones into the upper 32 bits. The contents of the Floating Point Register then look like a Quiet NaN.

Parameters

FRT Specifies target floating point register where result of operation is stored.

Examples

1. To load the contents of the Floating Point Status and Control Register into Floating Point Register 14, and fill the upper 32 bits of that register with ones:

```
# Assume FPSCR contains 0x0000 0000.
mffs 14
# FPR 14 now contains 0xFFFF FFFF 0000 0000.
```

Implementation Specifics

This instruction is part of Application Development Toolkit in AIX Base.

Related Information

Understanding The Floating Point Status and Control Register on page 1–12.

Understanding Floating Point Status and Control Register Instructions on page 1–14.

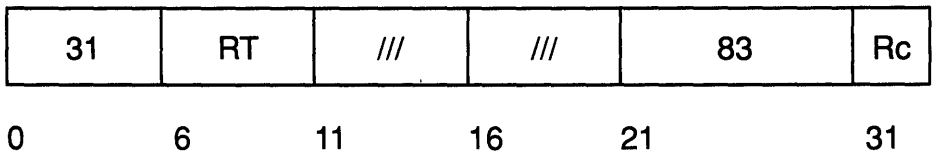
mfmsr (Move From Machine State Register) Instruction

Purpose

Copies the contents of the Machine State Register into a general purpose register.

Syntax

mfmsr *RT*



Description

The **mfmsr** instruction copies the contents of the Machine State Register into the target General Purpose Register *RT*.

The **mfmsr** instruction has one syntax form and does not affect the Fixed Point Exception Register or Condition Register Field 0.

Parameters

RT Specifies target general purpose register where result of operation is stored.

Examples

1. To copy the contents of the Machine State Register into GPR 4:

```
mfmsr 4
# GPR 4 now holds a copy of the bit
# settings of the Machine State Register.
```

Implementation Specifics

This instruction is part of Application Development Toolkit in AIX Base.

Related Information

Understanding Fixed Point Move To/From System Registers Instructions on page 1–11.

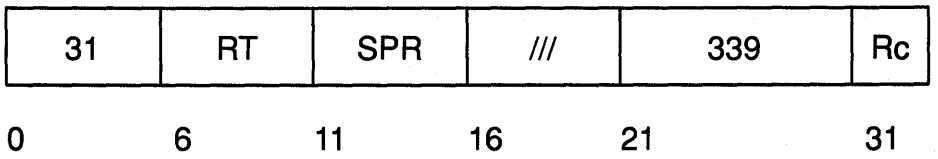
mfspr (Move From Special Purpose Register) Instruction

Purpose

Copies the contents of a special purpose register into a general purpose register.

Syntax

mfspr *RT,SPR*



Description

The **mfspr** instruction copies the contents of the Special Purpose Register *SPR* into the target General Purpose Register *RT*.

The Special Purpose Register identifier *SPR* can have any of the values specified in the following table.

SPR	Register
00000	MQ
00001	XER
00100	RTCU
00101	RTCL
00110	DEC
01000	LR
01001	CTR

All other combinations are reserved.

The **mfspr** instruction has one syntax form and does not affect the Fixed Point Exception Register or Condition Register Field 0.

Parameters

RT Specifies target general purpose register where result of operation is stored.

SPR Specifies source special purpose register for operation.

Extended Mnemonics

Seven extended mnemonic move from special purpose register instructions are based on the **mfspr** (Move from Special Purpose Register) instruction. The second parameter is encoded in the instruction, and the *RT* field is moved to the encoded special purpose register. The programmer explicitly specifies *RT*, and the extended mnemonic specifies the special purpose register.

Syntax	Parameters	Description
mfmq	*RT*	Move from MQ
mfxer	*RT*	Move from XER
mfrtcu	*RT*	Move from RTCU
mfrtcl	*RT*	Move from RTCL
mfdec	*RT*	Move from DEC
mflr	*RT*	Move from LR
mfctr	*RT*	Move from CTR

Examples

1. To copy the contents of the Fixed Point Exception Register into GPR 6:

```
mfspr 6,1
# GPR 6 now contains the bit settings of the Fixed
# Point Exception Register.
```

Implementation Specifics

This instruction is part of Application Development Toolkit in AIX Base.

Related Information

Understanding Fixed Point Move To/From System Registers Instructions on page 1–11.

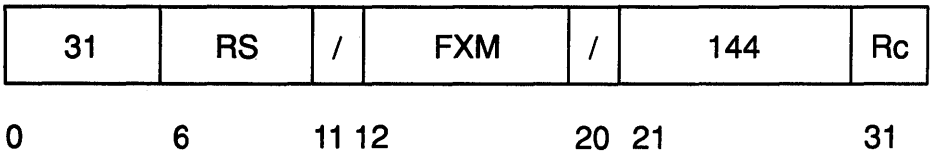
mtcrf (Move To Condition Register Fields) Instruction

Purpose

Copies the contents of a general purpose register into the Condition Register under control of a field mask.

Syntax

mtcrf *FXM,RS*



Description

The **mtcrf** instruction copies the contents of the source General Purpose Register *RS* into the Condition Register under the control of the field mask *FXM*.

The field mask *FXM* is defined as follows:

Bit	Description
12	CR 00–03 is updated with the contents of *RS* 00–03.
13	CR 04–07 is updated with the contents of *RS* 04–07.
14	CR 08–11 is updated with the contents of *RS* 08–11.
15	CR 12–15 is updated with the contents of *RS* 12–15.
16	CR 16–19 is updated with the contents of *RS* 16–19.
17	CR 20–23 is updated with the contents of *RS* 20–23.
18	CR 24–27 is updated with the contents of *RS* 24–27.
19	CR 28–31 is updated with the contents of *RS* 28–31.

The **mtcrf** instruction has one syntax form and does not affect the Fixed Point Exception Register.

Parameters

FXM	Specifies field mask.
RS	Specifies source general purpose register for operation.

Extended Mnemonics

One extended mnemonic move instruction is based on the **mtcrf** (Move to Condition Register Fields) instruction. It has the same effect as coding the following example.

```
mtcrf 0xff,RS
```

Syntax	Parameters	Description
mtcf	*RS*	Moves the contents of *RS* into the Condition Register.

Examples

1. To copy bits 00–03 of GPR 5 into Condition Register Field 0:

```
# Assume GPR 5 contains 0x7542 FFEE.
# Use the mask for Condition Register
# Field 0 (0x80 = b'1000 0000').
mtcrf 0x80,5
# Condition Register Field 0 now contains b'0111'.
```

Implementation Specifics

This instruction is part of Application Development Toolkit in AIX Base.

Related Information

See the following article in *POWERstation and POWERserver Hardware Technical Reference — General Information*: Condition Register.

Understanding Fixed Point Move To/From System Registers Instructions on page 1–11.

mtfsf (Move To FPSCR Fields) Instruction

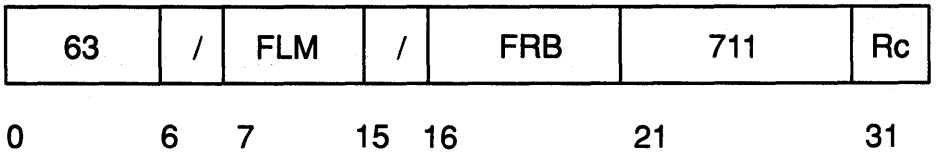
Purpose

Copies the contents of a Floating Point Register into the Floating Point Status and Control Register under the control of a field mask.

Syntax

mtfsf *FLM,FRB*

mtfsf. *FLM,FRB*



Description

The **mtfsf** instruction copies bits 32–63 of the contents of the Floating Point Register *FRB* into the Floating Point Status and Control Register under the control of the field mask specified by *FLM*.

The field mask *FLM* is defined as follows:

Bit	Description
7	FPSCR 00–03 is updated with the contents of *FRB* 32–35.
8	FPSCR 04–07 is updated with the contents of *FRB* 36–39.
9	FPSCR 08–11 is updated with the contents of *FRB* 40–43.
10	FPSCR 12–15 is updated with the contents of *FRB* 44–47.
11	FPSCR 16–19 is updated with the contents of *FRB* 48–51.
12	FPSCR 20–23 is updated with the contents of *FRB* 52–55.
13	FPSCR 24–27 is updated with the contents of *FRB* 56–59.
14	FPSCR 28–31 is updated with the contents of *FRB* 60–63.

The **mtfsf** instruction has two syntax forms. Each syntax form has a different effect on Condition Register Field 1.

Syntax form	FPSCR bits	Record bit (Rc)	Condition Register Field 1
mtfsf	None	0	None
mtfsf.	None	1	FX, FEX, VX, OX

The two syntax forms of the **mtfsf** instruction never affect the Fixed Point Exception Register. If the syntax form sets the Record (Rc) bit to 1, the instruction affects the Floating Point Exception (FX), Floating Point Enabled Exception (FEX), Floating Invalid Operation Exception (VX), and Floating Point Overflow Exception (OX) bits in Condition Register Field 1.

Note: When specifying FPSCR 0–3, some bits cannot be explicitly set or reset.

Parameters

FLM	Specifies field mask.
FRB	Specifies source floating point register for operation.

Extended Mnemonics

Two extended mnemonic move instructions are based on the **mtfsf** (Move to FPSCR Fields) instruction. They have the same effect as coding the following example.

```
mtfsf  0xff,FRB
mtfsf. 0xff,FRB
```

Syntax	Parameters	Description
mtfs	*FRB*	Moves the contents of *FRB* into the *FPSCR*.
mtfs.	*FRB*	Moves the contents of *FRB* into the *FPSCR*.

Examples

1. To copy the contents of Floating Point Register 5 bits 32–35 into Floating Point Status and Control Register Field 0:

```
# Assume bits 32–63 of Floating Point Register 5
# contain 0x3000 3000.
mtfsf 0x80,5
# Floating Point Status and Control Register
# Field 0 is set to b'0001'.
```

2. To copy the contents of Floating Point Register 5 bits 32–43 into Floating Point Status and Control Register Fields 0–2 and set Condition Register Field 1 to reflect the result of the operation:

```
# Assume bits 32–63 of Floating Point Register 5
# contains 0x2320 0000.
mtfsf. 0xE0,5
# Floating Point Status and Control Register Fields 0–2
# now contain b'0010 0011 0010'.
# Condition Register Field 1 now contains 0x2.
```

Implementation Specifics

This instruction is part of Application Development Toolkit in AIX Base.

Related Information

Understanding The Floating Point Status and Control Register on page 1–12.

mtfsfi (Move To FPSCR Field Immediate) Instruction

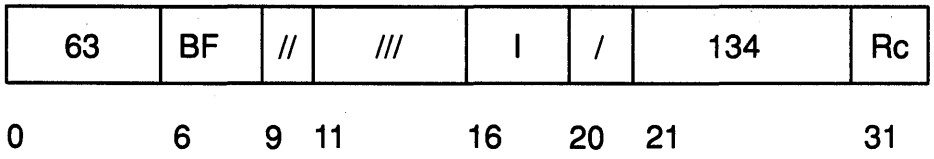
Purpose

Copies an immediate value into a specified Floating Point Status and Control Register field.

Syntax

mtfsfi *BF,I*

mtfsfi. *BF,I*



Description

The **mtfsfi** instruction copies the specified immediate value *I* into the Floating Point Status and Control Register field specified by *BF*. None of the other fields of the Floating Point Status and Control Register are affected.

The **mtfsfi** instruction has two syntax forms. Each syntax form has a different effect on Condition Register Field 1.

Syntax form	FPSCR bits	Record bit (Rc)	Condition Register Field 1
mtfsfi	None	0	None
mtfsfi.	None	1	FX, FEX, VX, OX

The two syntax forms of the **mtfsfi** instruction never affect the Floating Point Status and Control Register fields. If the syntax form sets the Record (Rc) bit to 1, the instruction affects the Floating Point Exception (FX), Floating Point Enabled Exception (FEX), Floating Invalid Operation Exception (VX), and Floating Point Overflow Exception (OX) bits in Condition Register Field 1.

Note: When specifying FPSCR 0–3, some bits cannot be explicitly set or reset.

Parameters

BF Specifies target Floating Point Status and Control Register field for operation.

I Specifies source immediate value for operation.

Examples

1. To set Floating Point Status and Control Register Field 6 to b'0100':

```
mtfsfi 6,4
# Floating Point Status and Control Register Field 6
# is now b'0100'.
```

2. To set Floating Point Status and Control Register field 0 to b'0100' and set Condition Register Field 1 to reflect the result of the operation:

```
mtfsfi. 0,1
# Floating Point Status and Control Register Field 0
# is now b'0001'.
# Condition Register Field 1 now contains 0x1.
```

Implementation Specifics

This instruction is part of Application Development Toolkit in AIX Base.

Related Information

Understanding The Floating Point Status and Control Register on page 1–12.

mtfsb1 (Move To FPSCR Bit 1) Instruction

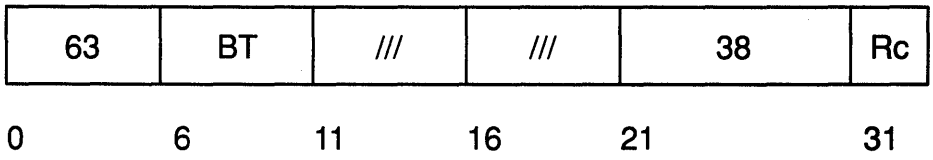
Purpose

Sets a specified Floating Point Status and Control Register bit to one.

Syntax

mtfsb1 *BT*

mtfsb1. *BT*



Description

The **mtfsb1** instruction sets the Floating Point Status and Control Register (FPSCR) bit specified by *BT* to one.

The **mtfsb1** instruction has two syntax forms. Each syntax form has a different effect on Condition Register Field 0.

Syntax form	FPSCR bits	Record bit (Rc)	Condition Register Field 1
mtfsb1	None	0	None
mtfsb1.	None	1	FX, FEX, VX, OX

The two syntax forms of the **mtfsb1** instruction never affect the Fixed Point Exception Register. If the syntax form sets the Record (Rc) bit to 1, the instruction affects the Floating Point Exception (FX), Floating Point Enabled Exception (FEX), Floating Invalid Operation Exception (VX), and Floating Point Overflow Exception (OX) bits in Condition Register Field 1.

Note: Bits 1–2 cannot be explicitly set or reset.

Parameters

BT Specifies FPSCR bit set to one by instruction.

Examples

1. To set the Floating Point Status and Control Register bit 4 to one:

```
mtfsb1 4
# Now bit 4 of the Floating Point Status and Control
# Register is set to 1.
```

2. To set the Floating Point Status and Control Register Overflow Exception Bit (bit 3) to one and set Condition Register Field 1 to reflect the result of the operation:

```
mtfsb1. 3
# Now bit 3 of the Floating Point Status and Control
# Register is set to 1.
```

Implementation Specifics

This instruction is part of Application Development Toolkit in AIX Base.

Related Information

Understanding The Floating Point Status and Control Register on page 1–12.

mtfsb0 (Move To FPSCR Bit 0) Instruction

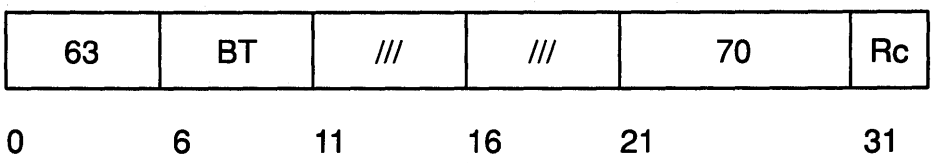
Purpose

Sets a specified Floating Point Status and Control Register bit to zero.

Syntax

mtfsb0 *BT*

mtfsb0. *BT*



Description

The **mtfsb0** instruction sets the Floating Point Status and Control Register bit specified by *BT* to zero.

The **mtfsb0** instruction has two syntax forms. Each syntax form has a different effect on Condition Register Field 0.

Syntax form	Fixed Point Exception Register	Record bit (Rc)	Condition Register Field 1
mtfsb0	None	0	None
mtfsb0.	None	1	FX, FEX, VX, OX

The two syntax forms of the **mtfsb0** instruction never affect the Fixed Point Exception Register. If the syntax form sets the Record (Rc) bit to 1, the instruction affects the Floating Point Exception (FX), Floating Point Enabled Exception (FEX), Floating Invalid Operation Exception (VX), and Floating Point Overflow Exception (OX) bits in Condition Register Field 1.

Note: Bits 1–2 cannot be explicitly set or reset.

Parameters

BT Specifies Floating Point Status and Control Register bit set by operation.

Examples

1. To set the Floating Point Status and Control Register Floating Point Overflow Exception Bit (bit 3) to zero:

```
mtfsb0 3
# Now bit 3 of the Floating Point Status and Control
# Register is 0.
```

2. To set the Floating Point Status and Control Register Floating Point Overflow Exception Bit (bit 3) to zero and set Condition Register Field 1 to reflect the result of the operation:

```
mtfsb0. 3
# Now bit 3 of the Floating Point Status and Control
# Register is 0.
```

Implementation Specifics

This instruction is part of Application Development Toolkit in AIX Base.

Related Information

Understanding The Floating Point Status and Control Register on page 1–12.

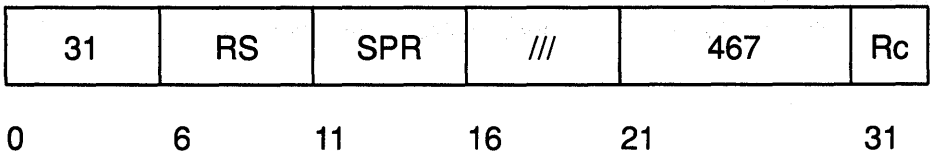
mtspr (Move To Special Purpose Register) Instruction

Purpose

Copies the contents of a general purpose register into a special purpose register.

Syntax

mtspr *SPR,RS*



Description

The **mtspr** instruction copies the contents of the source General Purpose Register *RS* into the target Special Purpose Register *SPR*.

The Special Purpose Register identifier *SPR* can have any of the values specified in the following table.

SPR	Register
00000	MQ
00001	XER
01000	LR
01001	CTR

All other combinations are reserved.

The **mtspr** instruction has one syntax form and does not affect the Fixed Point Exception Register or Condition Register Field 0.

Parameters

SPR Specifies target special purpose register for operation.

RS Specifies source general purpose register for operation.

Extended Mnemonics

Four extended mnemonic move from special purpose register instructions are based on the **mtspr** (Move to Special Purpose Register) instruction. The second parameter is encoded in the instruction, and the *RT* field is moved to the encoded special purpose register. The programmer explicitly specifies *RT*, and the extended mnemonic specifies the special purpose register.

Syntax	Parameters	Description
mtmq	*RT*	Move to MQ
mtxer	*RT*	Move to XER
mtlr	*RT*	Move to LR
mtctr	*RT*	Move to CTR

Examples

1. To copy the contents of GPR 5 into the Link Register:

```
# Assume GPR 5 holds 0x1000 00FF.
mtspr 8,5
# The Link Register now holds 0x1000 00FF.
```

Implementation Specifics

This instruction is part of Application Development Toolkit in AIX Base.

Related Information

Understanding Fixed Point Move To/From System Registers Instructions on page 1–11.

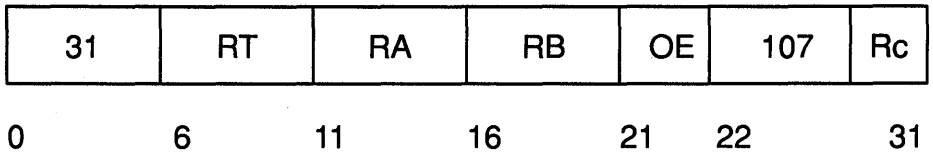
mul (Multiply) Instruction

Purpose

Multiplies the contents of two general purpose registers and stores the result in a general purpose register.

Syntax

mul *RT,RA,RB*
mul. *RT,RA,RB*
mulo *RT,RA,RB*
mulo. *RT,RA,RB*



Description

The **mul** instruction multiplies the contents of General Purpose Register *RA* and General Purpose Register *RB* and stores bits 0–31 of the result in the target General Purpose Register *RT* and bits 32–63 of the result in the MQ Register.

The **mul** instruction has four syntax forms. Each syntax form has a different effect on Condition Register Field 0 and the Fixed Point Exception Register.

Syntax form	Overflow Exception (OE)	Fixed Point Exception Register	Record bit (Rc)	Condition Register Field 0
mul	0	None	0	None
mul.	0	None	1	LT,GT,EQ
mulo	1	SO,OV	0	None
mulo.	1	SO,OV	1	LT,GT,EQ

The four syntax forms of the **mul** instruction never affect the Carry bit (CA) in the Fixed Point Exception Register. If the syntax form sets the Overflow Exception (OE) bit to 1, the instruction sets the Summary Overflow (SO) and Overflow (OV) bits in the Fixed Point Exception Register to 1 if the product is greater than 32 bits. If the syntax form sets the Record (Rc) bit to 1, then the Less Than (LT) zero, Greater Than (GT) zero and Equal To (EQ) zero bits in Condition Register Field 0 reflect the result in the low order 32 bits of the MQ Register.

Parameters

RT Specifies target general purpose register where result of operation is stored.

RA Specifies source general purpose register for operation.

RB Specifies source general purpose register for operation.

Examples

1. To multiply the contents of GPR 4 by the contents of GPR 10 and store the result in GPR 6 and the MQ register:

```
# Assume GPR 4 contains 0x0000 0003.
# Assume GPR 10 contains 0x0000 0002.
mul 6,4,10
# MQ register now contains 0x0000 0006.
# GPR 6 now contains 0x0000 0000.
```

2. To multiply the contents of GPR 4 by the contents of GPR 10, store the result in GPR 6 and the MQ register, and set Condition Register Field 0 to reflect the result of the operation:

```
# Assume GPR 4 contains 0x0000 4500.
# Assume GPR 10 contains 0x8000 7000.
mul. 6,4,10
# MQ register now contains 0x1E30 0000.
# GPR 6 now contains 0xFFFF DD80.
# Condition Register Field 0 now contains 0x4.
```

3. To multiply the contents of GPR 4 by the contents of GPR 10, store the result in GPR 6 and the MQ register, and set the Summary Overflow and Overflow bits in the Fixed Point Exception Register to reflect the result of the operation:

```
# Assume GPR 4 contains 0x0000 4500.
# Assume GPR 10 contains 0x8000 7000.
# Assume XER = 0.
mulo 6,4,10
# MQ register now contains 0x1E30 0000.
# GPR 6 now contains 0xFFFF DD80.
# XER now contains 0xc000 0000.
```

4. To multiply the contents of GPR 4 by the contents of GPR 10, store the result in GPR 6 and the MQ register, and set the Summary Overflow, Overflow, and Carry bits in the Fixed Point Exception Register and Condition Register Field 0 to reflect the result of the operation:

```
# Assume GPR 4 contains 0x0000 4500.
# Assume GPR 10 contains 0x8000 7000.
# Assume XER = 0.
mulo. 6,4,10
# MQ register now contains 0x1E30 0000.
# GPR 6 now contains 0xFFFF DD80.
# Condition Register Field 0 now contains 0x5.
# XER now contains 0xc000 0000.
```

Implementation Specifics

This instruction is part of Application Development Toolkit in AIX Base.

Related Information

Understanding Fixed Point Arithmetic Instructions on page 1–8.

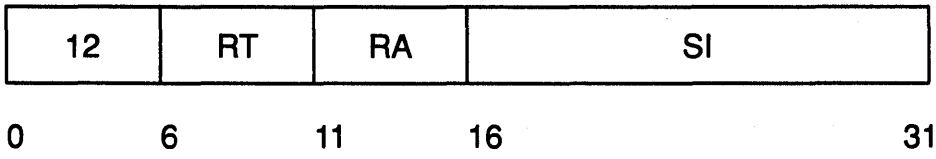
muli (Multiply Immediate) Instruction

Purpose

Multiplies the contents of a general purpose register by a 16-bit signed integer and stores the result in a general purpose register.

Syntax

muli *RT,RA,SI*



Description

The **muli** instruction multiplies the contents of General Purpose Register *RA* by a 16-bit signed integer and stores bits 32–63 of the result in the target General Purpose Register *RT*. The contents of the MQ Register are undefined.

The **muli** instruction has one syntax form and does not affect Condition Register Field 0 or the Fixed Point Exception Register.

Parameters

RT	Specifies target general purpose register where result of operation is stored.
RA	Specifies source general purpose register for operation.
SI	Specifies 16–bit signed integer for operation.

Examples

1. To multiply the contents of GPR 4 by 10 and place the result in GPR 6:

```
# Assume GPR 4 holds 0x0000 3000.
muli 6,4,10
# GPR 6 now holds 0x0001 E000.
```

Implementation Specifics

This instruction is part of Application Development Toolkit in AIX Base.

Related Information

Understanding Fixed Point Arithmetic Instructions on page 1–8.

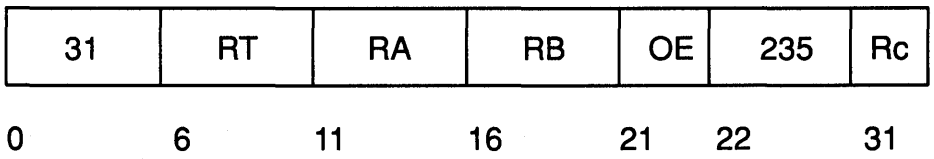
muls (Multiply Short) Instruction

Purpose

Multiplies the contents of two general purpose registers and stores the result in a general purpose register.

Syntax

muls *RT,RA,RB*
muls. *RT,RA,RB*
mulso *RT,RA,RB*
mulso. *RT,RA,RB*



Description

The **muls** intruction multiplies the contents of General Purpose Register *RA* and General Purpose Register *RB* and stores bits 32–63 of the result in the target General Purpose Register *RT*. The contents of the MQ Register are undefined.

The **muls** instruction has four syntax forms. Each syntax form has a different effect on Condition Register Field 0 and the Fixed Point Exception Register.

Syntax form	Overflow Exception (OE)	Fixed Point Exception Register	Record bit (Rc)	Condition Register Field 0
muls	0	None	0	None
muls.	0	None	1	LT,GT,EQ
mulso	1	SO,OV	0	None
mulso.	1	SO,OV	1	LT,GT,EQ

The four syntax forms of the **muls** instruction never affect the Carry bit (CA) in the Fixed Point Exception Register. If the syntax form sets the Overflow Exception (OE) bit to 1, the instruction sets the Summary Overflow (SO) and Overflow (OV) bits in the Fixed Point Exception Register to 1 if the product is greater than 32 bits. If the syntax form sets the Record (Rc) bit to 1, then the Less Than (LT) zero, Greater Than (GT) zero, and Equal To (EQ) zero bits in Condition Register Field 0 reflect the result in the low order 32 bits of the MQ Register.

Parameters

RT Specifies target general purpose register where result of operation is stored.

RA Specifies source general purpose register for operation.

RB Specifies source general purpose register for operation.

Examples

1. To multiply the contents of GPR 4 by the contents of GPR 10 and store the result in GPR 6:

```
# Assume GPR 4 holds 0x0000 3000.
# Assume GPR 10 holds 0x0000 7000.
muls 6,4,10
# GPR 6 now holds 0x1500 0000.
```

2. To multiply the contents of GPR 4 by the contents of GPR 10, store the result in GPR 6, and set and set Condition Register Field 0 to reflect the result of the operation:

```
# Assume GPR 4 holds 0x0000 4500.
# Assume GPR 10 holds 0x0000 7000.
muls. 6,4,10
# GPR 6 now holds 0x1E30 0000.
# Condition Register Field 0 now contains 0x4.
```

3. To multiply the contents of GPR 4 by the contents of GPR 10, store the result in GPR 6, and set the Summary Overflow and Overflow bits in the Fixed Point Exception Register to reflect the result of the operation:

```
# Assume GPR 4 holds 0x0000 4500.
# Assume GPR 10 holds 0x0007 0000.
# Assume XER = 0.
mulso 6,4,10
# GPR 6 now holds 0xE300 0000.
# XER now contains 0xc000 0000
```

4. To multiply the contents of GPR 4 by the contents of GPR 10, store the result in GPR 6, and set the Summary Overflow, Overflow, and Carry bits in the Fixed Point Exception Register and Condition Register Field 0 to reflect the result of the operation:

```
# Assume GPR 4 holds 0x0000 4500.
# Assume GPR 10 holds 0x7FFF FFFF.
# Assume XER = 0.
mulso. 6,4,10
# GPR 6 now holds 0xFFFF BB00.
# XER now contains 0xc000 0000
# Condition Register Field 0 now contains 0x9.
```

Implementation Specifics

This instruction is part of Application Development Toolkit in AIX Base.

Related Information

Understanding Fixed Point Arithmetic Instructions on page 1–8.

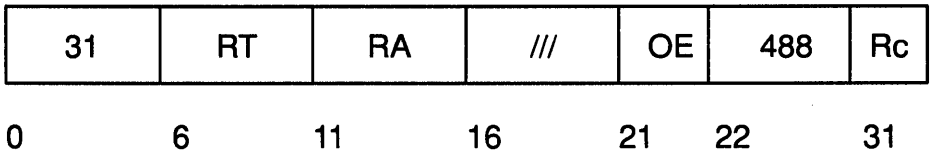
nabs (Negative Absolute) Instruction

Purpose

Negates the absolute value of the contents of a general purpose register and stores the result in a general purpose register.

Syntax

nabs *RT,RA*
nabs. *RT,RA*
nabso *RT,RA*
nabso. *RT,RA*



Description

The **nabs** instruction places the negative absolute value of the contents of General Purpose Register *RA* into the target General Purpose Register *RT*.

The **nabs** instruction has four syntax forms. Each syntax form has a different effect on Condition Register Field 0 and the Fixed Point Exception Register.

Syntax form	Overflow Exception (OE)	Fixed Point Exception Register	Record bit (Rc)	Condition Register Field 0
nabs	0	None	0	None
nabs.	0	None	1	LT,GT,EQ,SO
nabso	1	SO,OV	0	None
nabso.	1	SO,OV	1	LT,GT,EQ,SO

The four syntax forms of the **nabs** instruction never affect the Carry bit (CA) in the Fixed Point Exception Register. If the syntax form sets the Overflow Exception (OE) bit to 1, the Summary Overflow (SO) bit is unchanged and the Overflow (OV) bit is set to zero. If the syntax form sets the Record (Rc) bit to 1, the instruction affects the Less Than (LT) zero, Greater Than (GT) zero, Equal To (EQ) zero, and Summary Overflow (SO) bits in Condition Register Field 0.

Parameters

RT Specifies target general purpose register where result of operation is stored.

RA Specifies source general purpose register for operation.

Examples

1. To take the negative absolute value of the contents of GPR 4 and store the result in GPR 6:

```
# Assume GPR 4 contains 0x0000 3000.
nabs 6,4
# GPR 6 now contains 0xFFFF D000.
```

2. To take the negative absolute value of the contents of GPR 4, store the result in GPR 6, and set Condition Register Field 0 to reflect the result of the operation:

```
# Assume GPR 4 contains 0xFFFF FFFF.
nabs. 6,4
# GPR 6 now contains 0xFFFF FFFF.
```

3. To take the negative absolute value of the contents of GPR 4, store the result in GPR 6, and set the Overflow bit in the Fixed Point Exception Register to 0:

```
# Assume GPR 4 contains 0x0000 0001.
nabso 6,4
# GPR 6 now contains 0xFFFF FFFF.
```

4. To take the negative absolute value of the contents of GPR 4, store the result in GPR 6, set Condition Register Filed 0 to reflect the result of the operation, and set the Overflow bit in the Fixed Point Exception Register to 0:

```
# Assume GPR 4 contains 0x8000 0000.
nabso 6,4
# GPR 6 now contains 0x8000 0000.
```

Implementation Specifics

This instruction is part of Application Development Toolkit in AIX Base.

Related Information

Understanding Fixed Point Arithmetic Instructions on page 1–8.

nand (NAND) Instruction

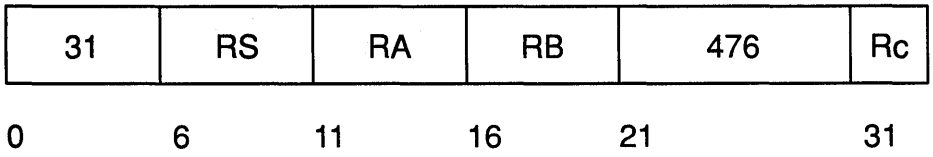
Purpose

Logically complements the result of ANDing the contents of two general purpose registers and stores the result in a general purpose register.

Syntax

nand *RA,RS,RB*

nand. *RA,RS,RB*



Description

The **nand** instruction ANDs the contents of General Purpose Register *RS* with the contents of General Purpose Register *RB* and stores the complement of the result in the target General Purpose Register *RA*.

The **nand** instruction has two syntax forms. Each syntax form has a different effect on Condition Register Field 0.

Syntax form	Overflow Exception (OE)	Fixed Point Exception Register	Record bit (Rc)	Condition Register Field 0
nand	None	None	0	None
nand.	None	None	1	LT,GT,EQ,SO

The two syntax forms of the **nand** instruction never affect the Fixed Point Exception Register. If the syntax form sets the Record (Rc) bit to 1, the instruction affects the Less Than (LT) zero, Greater Than (GT) zero, Equal To (EQ) zero, and Summary Overflow (SO) bits in Condition Register Field 0.

Parameters

RA Specifies target general purpose register where result of operation is stored.

RS Specifies source general purpose register for operation.

RB Specifies source general purpose register for operation.

Examples

1. To complement the result of ANDing the contents of GPR 4 and GPR 7 and store the result in GPR 6:

```
# Assume GPR 4 contains 0x9000 3000.
# Assume GPR 7 contains 0x789A 789B.
nand 6,4,7
# GPR 7 now contains 0xEFFF CFFF.
```

2. To complement the result of ANDing the contents of GPR 4 and GPR 7, store the result in GPR 6, and set Condition Register Field 0 to reflect the result of the operation:

```
# Assume GPR 4 contains 0xB004 3000.
# Assume GPR 7 contains 0x789A 789B.
nand. 6,4,7
# GPR 6 now contains 0xCFFF CFFF.
```

Implementation Specifics

This instruction is part of Application Development Toolkit in AIX Base.

Related Information

Understanding Fixed Point Logical Instructions on page 1–9.

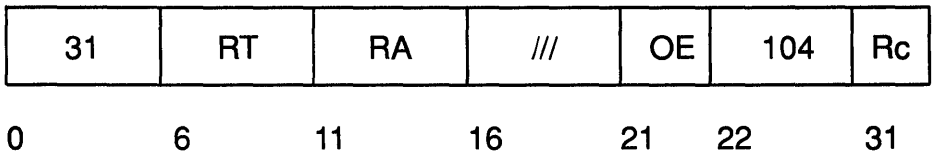
neg (Negate) Instruction

Purpose

Changes the arithmetic sign of the contents of a general purpose register and places the result in a general purpose register.

Syntax

neg *RT,RA*
neg. *RT,RA*
nego *RT,RA*
nego. *RT,RA*



Description

The **neg** instruction adds 1 to the one's complement of the contents of a General Purpose Register *RA* and stores the result in General Purpose Register *RT*.

If General Purpose Register *RA* contains the most negative number (i.e., 0x8000 0000), the result of the instruction is the most negative number and signals the Overflow bit in the Fixed Point Exception Register if OE is 1.

The **neg** instruction has four syntax forms. Each syntax form has a different effect on Condition Register Field 0 and the Fixed Point Exception Register.

Syntax form	Overflow Exception (OE)	Fixed Point Exception Register	Record bit (Rc)	Condition Register Field 0
neg	0	None	0	None
neg.	0	None	1	LT,GT,EQ,SO
nego	1	SO,OV	0	None
nego.	1	SO,OV	1	LT,GT,EQ,SO

The four syntax forms of the **neg** instruction never affect the Carry bit (CA) in the Fixed Point Exception Register. If the syntax form sets the Overflow Exception (OE) bit to 1, the instruction affects the Summary Overflow (SO) and Overflow (OV) bits in the Fixed Point Exception Register. If the syntax form sets the Record (Rc) bit to 1, the instruction affects the Less Than (LT) zero, Greater Than (GT) zero, Equal To (EQ) zero, and Summary Overflow (SO) bits in Condition Register Field 0.

Parameters

RT Specifies target general purpose register where result of operation is stored.

RA Specifies source general purpose register for operation.

Examples

1. To negate the contents of GPR 4 and store the result in GPR 6:

```
# Assume GPR 4 contains 0x9000 3000.
neg 6,4
# GPR 6 now contains 0x6FFF D000.
```

2. To negate the contents of GPR 4, store the result in GPR 6, and set Condition Register Field 0 to reflect the result of the operation:

```
# Assume GPR 4 contains 0x789A 789B.
neg. 6,4
# GPR 6 now contains 0x8765 8765.
```

3. To negate the contents of GPR 4, store the result in GPR 6, and set the Fixed Point Exception Register Summary Overflow and Overflow bits to reflect the result of the operation:

```
# Assume GPR 4 contains 0x9000 3000.
nego 6,4
# GPR 6 now contains 0x6FFF D000.
```

4. To negate the contents of GPR 4, store the result in GPR 6, and set Condition Register Field 0 and the Fixed Point Exception Register Summary Overflow and Overflow bits to reflect the result of the operation:

```
# Assume GPR 4 contains 0x8000 0000.
nego. 6,4
# GPR 6 now contains 0x8000 0000.
```

Implementation Specifics

This instruction is part of Application Development Toolkit in AIX Base.

Related Information

Understanding Fixed Point Arithmetic Instructions on page 1–8.

nor (NOR) Instruction

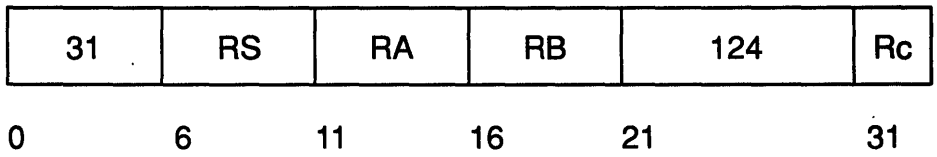
Purpose

Logically complements the result of ORing the contents of two general purpose registers and stores the result in a general purpose register.

Syntax

nor *RA,RS,RB*

nor. *RA,RS,RB*



Description

The **nor** instruction ORs the contents of General Purpose Register *RS* with the contents of General Purpose Register *RB* and stores the complemented result in General Purpose Register *RA*.

The **nor** instruction has two syntax forms. Each syntax form has a different effect on Condition Register Field 0.

Syntax form	Overflow Exception (OE)	Fixed Point Exception Register	Record bit (Rc)	Condition Register Field 0
nor	None	None	0	None
nor.	None	None	1	LT,GT,EQ,SO

The two syntax forms of the **nor** instruction never affect the Fixed Point Exception Register. If the syntax form sets the Record (Rc) bit to 1, the instruction affects the Less Than (LT) zero, Greater Than (GT) zero, Equal To (EQ) zero, and Summary Overflow (SO) bits in Condition Register Field 0.

Parameters

RA Specifies target general purpose register where result of operation is stored.

RS Specifies source general purpose register for operation.

RB Specifies source general purpose register for operation.

Examples

1. To NOR the contents of GPR 4 and GPR 7 and store the result in GPR 6:

```
# Assume GPR 4 contains 0x9000 3000.
# Assume GPR 7 contains 0x789A 789B.
nor 6,4,7
# GPR 7 now contains 0x0765 8764.
```

2. To NOR the contents of GPR 4 and GPR 7, store the result in GPR 6, and set Condition Register Field 0 to reflect the result of the operation:

```
# Assume GPR 4 contains 0xB004 3000.
# Assume GPR 7 contains 0x789A 789B.
nor. 6,4,7
# GPR 6 now contains 0x0761 8764.
```

Implementation Specifics

This instruction is part of Application Development Toolkit in AIX Base.

Related Information

Understanding Fixed Point Logical Instructions on page 1–9.

or (OR) Instruction

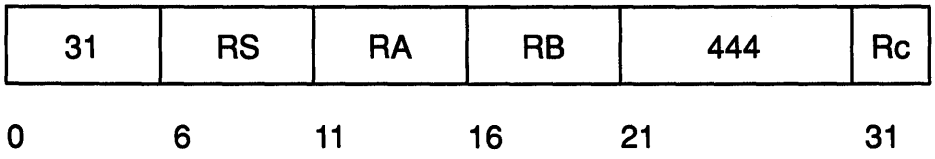
Purpose

Logically ORs the contents of two general purpose registers and stores the result in a general purpose register.

Syntax

or *RA,RS,RB*

or. *RA,RS,RB*



Description

The **or** instruction ORs the contents of General Purpose Register *RS* with the contents of General Purpose Register *RB* and stores the result in General Purpose Register *RA*.

The **or** instruction has two syntax forms. Each syntax form has a different effect on Condition Register Field 0.

Syntax form	Overflow Exception (OE)	Fixed Point Exception Register	Record bit (Rc)	Condition Register Field 0
or	None	None	0	None
or.	None	None	1	LT,GT,EQ,SO

The two syntax forms of the **or** instruction never affect the Fixed Point Exception Register. If the syntax form sets the Record (Rc) bit to 1, the instruction affects the Less Than (LT) zero, Greater Than (GT) zero, Equal To (EQ) zero, and Summary Overflow (SO) bits in Condition Register Field 0.

Parameters

RA Specifies target general purpose register where result of operation is stored.

RS Specifies source general purpose register for operation.

RB Specifies source general purpose register for operation.

Examples

1. To OR the contents of GPR 4 and GPR 7 and store the result in GPR 6:

```
# Assume GPR 4 contains 0x9000 3000.
# Assume GPR 7 contains 0x789A 789B.
or 6,4,7
# GPR 6 now contains 0xF89A 789B.
```

2. To OR the contents of GPR 4 and GPR 7, load the result in GPR 6, and set Condition Register Field 0 to reflect the result of the operation:

```
# Assume GPR 4 contains 0xB004 3000.
# Assume GPR 7 contains 0x789A 789B.
or. 6,4,7
# GPR 6 now contains 0xF89E 789B.
```

Implementation Specifics

This instruction is part of Application Development Toolkit in AIX Base.

Related Information

Understanding Fixed Point Logical Instructions on page 1–9.

orc (OR With Complement) Instruction

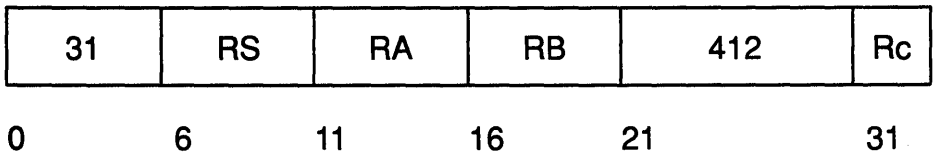
Purpose

Logically ORs the contents of a general purpose register with the complement of the contents of a general purpose register and stores the result in a general purpose register.

Syntax

orc *RA,RS,RB*

orc. *RA,RS,RB*



Description

The **orc** instruction ORs the contents of General Purpose Register *RS* with the complement of the contents of General Purpose Register *RB* and stores the result in General Purpose Register *RA*.

The **orc** instruction has two syntax forms. Each syntax form has a different effect on Condition Register Field 0.

Syntax form	Overflow Exception (OE)	Fixed Point Exception Register	Record bit (Rc)	Condition Register Field 0
orc	None	None	0	None
orc.	None	None	1	LT,GT,EQ,SO

The two syntax forms of the **orc** instruction never affect the Fixed Point Exception Register. If the syntax form sets the Record (Rc) bit to 1, the instruction affects the Less Than (LT) zero, Greater Than (GT) zero, Equal To (EQ) zero, and Summary Overflow (SO) bits in Condition Register Field 0.

Parameters

RA Specifies target general purpose register where result of operation is stored.

RS Specifies source general purpose register for operation.

RB Specifies source general purpose register for operation.

Examples

1. To OR the contents of GPR 4 with the complement of the contents of GPR 7 and store the result in GPR 6:

```
# Assume GPR 4 contains 0x9000 3000.
# Assume GPR 7 contains 0x789A 789B, whose
# complement is 0x8765 8764.
orc 6,4,7
# GPR 7 now contains 0x9765 B764.
```

2. To OR the contents of GPR 4 with the complement of the contents GPR 7, store the result in GPR 6, and set Condition Register Field 0 to reflect the result of the operation:

```
# Assume GPR 4 contains 0xB004 3000.
# Assume GPR 7 contains 0x789A 789B, whose
# complement is 0x8765 8764.
orc. 6,4,7
# GPR 7 now contains 0xB765 B764.
```

Implementation Specifics

This instruction is part of Application Development Toolkit in AIX Base.

Related Information

Understanding Fixed Point Logical Instructions on page 1–9.

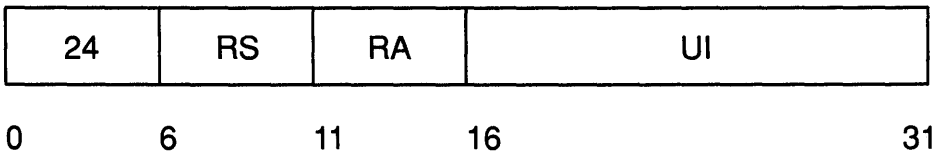
oril (OR Immediate Lower) Instruction

Purpose

OR's the lower 16 bits of the contents of a general purpose register with a 16-bit unsigned integer and stores the result in a general purpose register.

Syntax

oril *RA,RS,UI*



Description

The **oril** instruction ORs the contents of General Purpose Register *RS* with the concatenation of x'0000' and a 16-bit unsigned integer *UI* and places the result in General Purpose Register *RA*.

The **oril** instruction has one syntax form and does not affect Condition Register Field 0 or the Fixed Point Exception Register.

Parameters

RA	Specifies target general purpose register where result of operation is stored.
RS	Specifies source general purpose register for operation.
UI	Specifies 16–bit unsigned integer for operation.

Examples

1. To OR the lower 16 bits of the contents of GPR 4 with 0x0079 and store the result in GPR 6:

```
# Assume GPR 4 contains 0x9000 3000.
oril 6,4,0x0079
# GPR 6 now contains 0x9000 3079.
```

Implementation Specifics

This instruction is part of Application Development Toolkit in AIX Base.

Related Information

Understanding Fixed Point Logical Instructions on page 1–9.

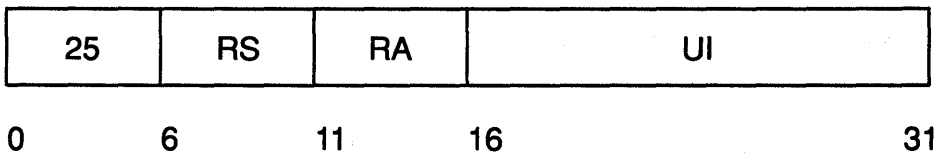
oriu (OR Immediate Upper) Instruction

Purpose

OR's the upper 16 bits of the contents of a general purpose register with a 16-bit unsigned integer and stores the result in a general purpose register.

Syntax

oriu *RA,RS,UI*



Description

The **oriu** instruction ORs the contents of General Purpose Register *RS* with the concatenation of a 16-bit unsigned integer *UI* and x'0000' and stores the result in General Purpose Register *RA*.

The **oriu** instruction has one syntax form and does not affect Condition Register Field 0 or the Fixed Point Exception Register.

Parameters

RA	Specifies target general purpose register where result of operation is stored.
RS	Specifies source general purpose register for operation.
UI	Specifies 16–bit unsigned integer for operation.

Examples

1. To OR the upper 16 bits of the contents of GPR 4 with 0x0079and store the result in GPR 6:

```
# Assume GPR 4 contains 0x9000 3000.
oriu 6,4,0x0079
# GPR 6 now contains 0x9079 3000.
```

Implementation Specifics

This instruction is part of Application Development Toolkit in AIX Base.

Related Information

Understanding Fixed Point Logical Instructions on page 1–9.

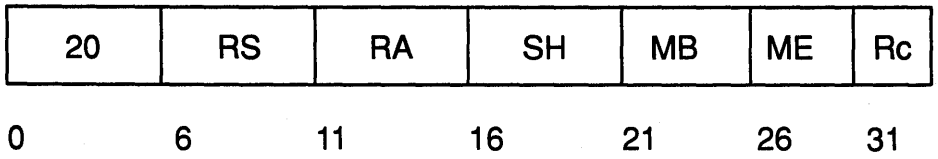
rlimi (Rotate Left Immediate Then Mask Insert) Instruction

Purpose

Rotates the contents of a general purpose register left by a specified number of bits and stores the result in a general purpose register under the control of a generated mask.

Syntax

rlimi *RA,RS,SH,MB,ME*
rlimi. *RA,RS,SH,MB,ME*
rlimi *RA,RS,SH,BM*
rlimi. *RA,RS,SH,BM*



Description

The **rlimi** instruction rotates the contents of the source General Purpose Register *RS* left SH bits and stores the rotated data in General Purpose Register *RA* under control of a 32–bit generated mask defined by the values in Mask Begin (*MB*) and Mask End (*ME*).

- If a mask bit is one, the instruction places the associated bit of rotated data in General Purpose Register *RA*; if a mask bit is zero, the General Purpose Register *RA* bit remains unchanged.
- If the *MB* value is less than the *ME* value + 1, then the mask bits between and including the starting point and the end point are set to ones. All other bits are set to zeros.
- If the *MB* value is the same as the *ME* value + 1, then all 32 mask bits are set to ones.
- If the *MB* value is greater than the *ME* value + 1, then all of the mask bits between and including the *ME* value +1 and the *MB* value –1 are set to zeros. All other bits are set to ones.

BM may also be used to specify the mask for this instruction. The assembler will generate the *MB* and *ME* parameters from *BM*.

The **rlimi** instruction has two syntax forms. Each syntax form has a different effect on Condition Register Field 0.

Syntax form	Overflow Exception (OE)	Fixed Point Exception Register	Record bit (Rc)	Condition Register Field 0
rlimi	None	None	0	None
rlimi.	None	None	1	LT,GT,EQ,SO

The two syntax forms of the **rlimi** instruction never affect the Fixed Point Exception Register. If the syntax form sets the Record (Rc) bit to 1, the instruction affects the Less Than (LT) zero, Greater Than (GT) zero, Equal To (EQ) zero, and Summary Overflow (SO) bits in Condition Register Field 0.

Parameters

RA	Specifies target general purpose register where result of operation is stored.
RS	Specifies source general purpose register for operation.
SH	Specifies shift value for operation.
MB	Specifies begin value of mask for operation.
ME	Specifies end value of mask for operation.
BM	Specifies value of 32–bit mask.

Examples

1. To rotate the contents of GPR 4 to the left 2 bits and store the masked result in GPR 6:

```
# Assume GPR 4 contains 0x9000 3000.
# Assume GPR 6 contains 0x0000 0003.
rlimi 6,4,2,0,0x1D
# GPR 6 now contains 0x4000 C003.
# Under the same conditions
#  rlimi 6,4,2,0xFFFFFFFC
# will produce the same result.
```

2. To rotate the contents of GPR 4 to the left 2 bits, store the masked result in GPR 6, and set Condition Register Field 0 to reflect the result of the operation:

```
# Assume GPR 4 contains 0x789A 789B.
# Assume GPR 6 contains 0x3000 0003.
rlimi. 6,4,2,0,0x1A
# GPR 6 now contains 0xE269 E263.
# CRF 0 now contains 0x8.
# Under the same conditions
#  rlimi. 6,4,2,0xFFFFFFE0
# will produce the same result.
```

Implementation Specifics

This instruction is part of Application Development Toolkit in AIX Base.

Related Information

Understanding Fixed Point Rotate Instructions on page 1–9.

rlinm (Rotate Left Immediate Then AND With Mask) Instruction

Purpose

ANDs a generated mask with the result of rotating the contents of a general purpose register left by a specified number of bits.

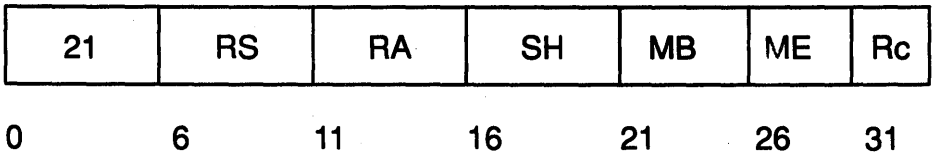
Syntax

rlinm *RA,RS,SH,MB,ME*

rlinm. *RA,RS,SH,MB,ME*

rlinm *RA,RS,SH,BM*

rlinm. *RA,RS,SH,BM*



Description

The **rlinm** instruction rotates the contents of the source General Purpose Register *RS* left *SH* bits, ANDs the rotated data with a 32–bit generated mask defined by the values in Mask Begin (*MB*) and Mask End (*ME*), and stores the result in General Purpose Register *RA*.

- If the *MB* value is less than the *ME* value + 1, then the mask bits between and including the starting point and the end point are set to ones. All other bits are set to zeros.
- If the *MB* value is the same as the *ME* value + 1, then all 32 mask bits are set to ones.
- If the *MB* value is greater than the *ME* value + 1, then all of the mask bits between and including the *ME* value +1 and the *MB* value –1 are set to zeros. All other bits are set to ones.

BM may also be used to specify the mask for this instruction. The assembler will generate the *MB* and *ME* parameters from *BM*.

The **rlinm** instruction has two syntax forms. Each syntax form has a different effect on Condition Register Field 0.

Syntax form	Overflow Exception (OE)	Fixed Point Exception Register	Record bit (Rc)	Condition Register Field 0
rlinm	None	None	0	None
rlinm.	None	None	1	LT,GT,EQ,SO

The two syntax forms of the **rlinm** instruction never affect the Fixed Point Exception Register. If the syntax form sets the Record (Rc) bit to 1, the instruction affects the Less Than (LT) zero, Greater Than (GT) zero, Equal To (EQ) zero, and Summary Overflow (SO) bits in Condition Register Field 0.

Parameters

RA Specifies target general purpose register where result of operation is stored.

RS	Specifies source general purpose register for operation.
SH	Specifies shift value for operation.
MB	Specifies begin value of mask for operation.
ME	Specifies end value of mask for operation.
BM	Specifies value of 32–bit mask.

Extended Mnemonics

Four extended mnemonic shift instructions are based on the **rlinm** (Rotate Left Immediate Then AND With Mask) instruction.

Syntax	Parameters	Description
sli	*RA,RS,SH*	Shifts *RS* to the left *SH* positions and places the result in *RA*.
sli.	*RA,RS,SH*	Shifts *RS* to the left *SH* positions, places the result in *RA*, and affects Condition Register Field 0.
sri	*RA,RS,SH*	Shifts *RS* to the right *SH* positions and places the result in *RA*.
sri.	*RA,RS,SH*	Shifts *RS* to the right *SH* positions, places the result in *RA*, and affects Condition Register Field 0.

Examples

1. To rotate the contents of GPR 4 to the left 2 bits and AND the result with a mask of 29 ones:

```
# Assume GPR 4 contains 0x9000 3000.
# Assume GPR 6 contains 0xFFFF FFFF.
rlinm 6,4,2,0,0x1D
# GPR 6 now contains 0x4000 C000.
# Under the same conditions
#  rlinm 6,4,2,0xFFFFFFFC
# will produce the same result.
```

2. To rotate the contents of GPR 4 to the left 2 bits, AND the result with a mask of 29 ones, and set Condition Register Field 0 to reflect the result of the operation:

```
# Assume GPR 4 contains 0xB004 3000.
# Assume GPR 6 contains 0xFFFF FFFF.
rlinm. 6,4,2,0,0x1D
# GPR 6 now contains 0xC010 C000.
# CRF 0 now contains 0x8.
# Under the same conditions
#  rlinm. 6,4,2,0xFFFFFFFC
# will produce the same result.
```

Implementation Specifics

This instruction is part of Application Development Toolkit in AIX Base.

Related Information

Understanding Fixed Point Rotate Instructions on page 1–9.

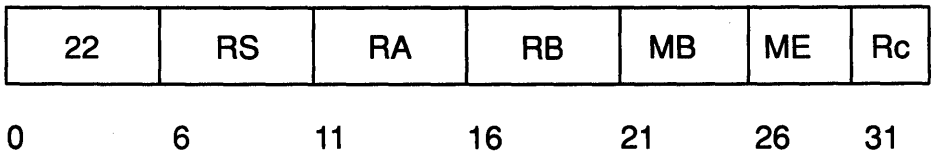
rlmi (Rotate Left Then Mask Insert) Instruction

Purpose

Rotates the contents of a general purpose register left by a number of bits in a general purpose register and stores the result in a general purpose register under the control of a generated mask.

Syntax

rlmi *RA,RS,RB,MB,ME*
rlmi. *RA,RS,RB,MB,ME*
rlmi *RA,RS,RB,BM*
rlmi. *RA,RS,RB,BM*



Description

The **rlmi** instruction rotates the contents of the source General Purpose Register *RS* left the number of bits specified by bits 27–31 of General Purpose Register *RB* and stores the rotated data in General Purpose Register *RA* under control of a 32–bit generated mask defined by the values in Mask Begin (*MB*) and Mask End (*ME*).

- If a mask bit is one, the instruction places the associated bit of rotated data in General Purpose Register *RA*; if a mask bit is zero, the General Purpose Register *RA* bit remains unchanged.
- If the *MB* value is less than the *ME* value + 1, then the mask bits between and including the starting point and the end point are set to ones. All other bits are set to zeros.
- If the *MB* value is the same as the *ME* value + 1, then all 32 mask bits are set to ones.
- If the *MB* value is greater than the *ME* value + 1, then all of the mask bits between and including the *ME* value +1 and the *MB* value –1 are set to zeros. All other bits are set to ones.

BM may also be used to specify the mask for this instruction. The assembler will generate the *MB* and *ME* parameters from *BM*.

The **rlmi** instruction has two syntax forms. Each syntax form has a different effect on Condition Register Field 0.

Syntax form	Overflow Exception (OE)	Fixed Point Exception Register	Record bit (Rc)	Condition Register Field 0
rlmi	None	None	0	None
rlmi.	None	None	1	LT,GT,EQ,SO

The two syntax forms of the **rlmi** instruction never affect the Fixed Point Exception Register. If the syntax form sets the Record (Rc) bit to 1, the instruction affects the Less Than (LT)

zero, Greater Than (GT) zero, Equal To (EQ) zero, and Summary Overflow (SO) bits in Condition Register Field 0.

Parameters

RA	Specifies target general purpose register where result of operation is stored.
RS	Specifies source general purpose register for operation.
RB	Specifies general purpose register that contains number of bits for rotation of data.
MB	Specifies begin value of mask for operation.
ME	Specifies end value of mask for operation.
BM	Specifies value of 32–bit mask.

Examples

1. To rotate the contents of GPR 4 by the value contained in bits 27–31 in GPR 5 and store the masked result in GPR 6:

```
# Assume GPR 4 contains 0x9000 3000.
# Assume GPR 5 contains 0x0000 0002.
# Assume GPR 6 contains 0xFFFF FFFF.
rlmi 6,4,5,0,0x1D
# GPR 6 now contains 0x4000 C003.
# Under the same conditions
#  rlmi 6,4,5,0xFFFFFFFC
# will produce the same result.
```

2. To rotate the contents of GPR 4 by the value contained in bits 27–31 in GPR 5, store the masked result in GPR 6, and set Condition Register Field 0 to reflect the result of the operation:

```
# Assume GPR 4 contains 0xB004 3000.
# Assume GPR 5 contains 0x0000 0002.
# GPR 6 is the target register and contains 0xFFFF FFFF.
rlmi. 6,4,5,0,0x1D
# GPR 6 now contains 0xC010 C003.
# CRF 0 now contains 0x8.
# Under the same conditions
#  rlmi. 6,4,5,0xFFFFFFFC
# will produce the same result.
```

Implementation Specifics

This instruction is part of Application Development Toolkit in AIX Base.

Related Information

Understanding Fixed Point Rotate Instructions on page 1–9.

rlnm (Rotate Left Then AND With Mask) Instruction

Purpose

Rotates the contents of a general purpose register left by a number of bits specified in a general purpose register, ANDs the the rotated data with the generated mask, and stores the result in a general purpose register.

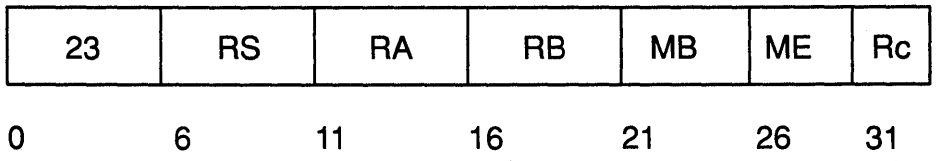
Syntax

rlnm *RA,RS,RB,MB,ME*

rlnm. *RA,RS,RB,MB,ME*

rlnm *RA,RS,SH,BM*

rlnm. *RA,RS,SH,BM*



Description

The **rlnm** instruction rotates the contents of the source General Purpose Register *RS* left the number of bits specified by bits 27–31 of General Purpose Register *RB*, ANDs the rotated data with a 32–bit generated mask defined by the values in Mask Begin (*MB*) and Mask End (*ME*), and stores the result in General Purpose Register *RA*.

- If the *MB* value is less than the *ME* value + 1, then the mask bits between and including the starting point and the end point are set to ones. All other bits are set to zeros.
- If the *MB* value is the same as the *ME* value + 1, then all 32 mask bits are set to ones.
- If the *MB* value is greater than the *ME* value + 1, then all of the mask bits between and including the *ME* value +1 and the *MB* value –1 are set to zeros. All other bits are set to ones.

BM may also be used to specify the mask for this instruction. The assembler will generate the *MB* and *ME* parameters from *BM*.

The **rlnm** instruction has two syntax forms. Each syntax form has a different effect on Condition Register Field 0.

Syntax form	Overflow Exception (OE)	Fixed Point Exception Register	Record bit (Rc)	Condition Register Field 0
rlnm	None	None	0	None
rlnm.	None	None	1	LT,GT,EQ,SO

The two syntax forms of the **rlnm** instruction never affect the Fixed Point Exception Register. If the syntax form sets the Record (Rc) bit to 1, the instruction affects the Less Than (LT) zero, Greater Than (GT) zero, Equal To (EQ) zero, and Summary Overflow (SO) bits in Condition Register Field 0.

Parameters

RA	Specifies target general purpose register where result of operation is stored.
RS	Specifies source general purpose register for operation.
RB	Specifies general purpose register that contains number of bits for rotation of data.
MB	Specifies begin value of mask for operation.
ME	Specifies end value of mask for operation.
BM	Specifies value of 32–bit mask.

Examples

1. To rotate the contents of GPR 4 two bits to the left, AND the result with a mask of 29 ones, and store the result in GPR 6:

```
# Assume GPR 4 contains 0x9000 3000.
# Assume GPR 5 contains 0x0000 0002.
# Assume GPR 6 contains 0xFFFF FFFF.
rlnm 6,4,5,0,0x1D
# GPR 6 now contains 0x4000 C000.
# Under the same conditions
#  rlnm 6,4,5,0xFFFFFFFC
# will produce the same result.
```

2. To rotate GPR 4 two bits to the left, AND the result with a mask of 29 ones, store the result in GPR 6, and set Condition Register Field 0 to reflect the result of the operation:

```
# Assume GPR 4 contains 0xB004 3000.
# Assume GPR 5 contains 0x0000 0002.
# Assume GPR 6 contains 0xFFFF FFFF.
rlnm. 6,4,5,0,0x1D
# GPR 6 now contains 0xC010 C000.
# CRF 0 now contains 0x8.
# Under the same conditions
#  rlnm. 6,4,5,0xFFFFFFFC
# will produce the same result.
```

Implementation Specifics

This instruction is part of Application Development Toolkit in AIX Base.

Related Information

Understanding Fixed Point Rotate Instructions on page 1–9.

rrib (Rotate Right And Insert Bit) Instruction

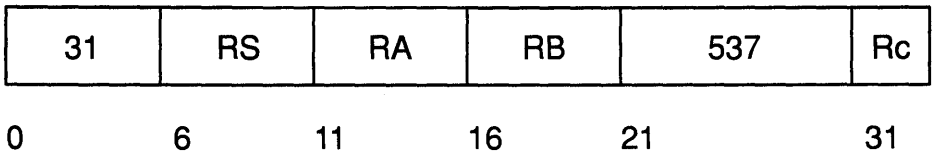
Purpose

Rotates bit 0 in a general purpose register right by a number of bits specified by a general purpose register and stores the rotated bit in a general purpose register.

Syntax

rrib *RA,RS,RB*

rrib. *RA,RS,RB*



Description

The **rrib** instruction rotates bit 0 of the source General Purpose Register *RS* right the number of bits specified by bits 27–31 of General Purpose Register *RB* and stores the rotated bit in General Purpose Register *RA*.

The **rrib** instruction has two syntax forms. Each syntax form has a different effect on Condition Register Field 0.

Syntax form	Overflow Exception (OE)	Fixed Point Exception Register	Record bit (Rc)	Condition Register Field 0
rrib	None	None	0	None
rrib.	None	None	1	LT,GT,EQ,SO

The two syntax forms of the **rrib** instruction never affect the Fixed Point Exception Register. If the syntax form sets the Record (Rc) bit to 1, the instruction affects the Less Than (LT) zero, Greater Than (GT) zero, Equal To (EQ) zero, and Summary Overflow (SO) bits in Condition Register Field 0.

Parameters

RA Specifies target general purpose register where result of operation is stored.

RS Specifies source general purpose register for operation.

RB Specifies general purpose register that contains the number of bits for rotation of data.

Examples

1. To rotate bit 0 of GPR 5 to the right 4 bits and store its value in GPR 4:

```
# Assume GPR 5 contains 0x0000 0000.
# Assume GPR 6 contains 0x0000 0004.
# Assume GPR 4 contains 0xFFFF FFFF.
rrib 4,5,6
# GPR 4 now contains 0xF7FF FFFF.
```

2. To rotate bit 0 of GPR 5 to the right 4 bits, store its value in GPR 4, and set Condition Register Field 0 to reflect the result of the operation:

```
# Assume GPR 5 contains 0xB004 3000.
# Assume GPR 6 contains 0x0000 0004.
# Assume GPR 4 contains 0x0000 0000.
rrib. 4,5,6
# GPR 4 now contains 0x0800 0000.
```

Implementation Specifics

This instruction is part of Application Development Toolkit in AIX Base.

Related Information

Understanding Fixed Point Rotate Instructions on page 1–9.

sf (Subtract From) Instruction

Purpose

Subtracts the contents of a general purpose register from the contents of a general purpose register and places the result in a general purpose register.

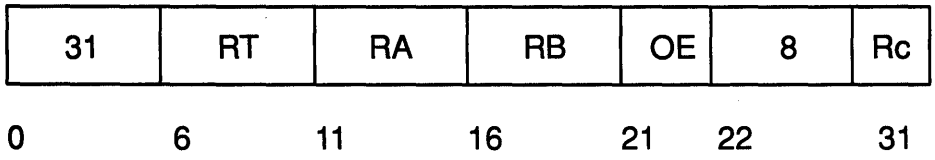
Syntax

sf *RT,RA,RB*

sf. *RT,RA,RB*

sfo *RT,RA,RB*

sfo. *RT,RA,RB*



Description

The **sf** instruction adds the ones complement of the contents of General Purpose Register *RA* and one to the contents of General Purpose Register *RB* and stores the result in the target General Purpose Register *RT*.

The **sf** instruction has four syntax forms. Each syntax form has a different effect on Condition Register Field 0 and the Fixed Point Exception Register.

Syntax form	Overflow Exception (OE)	Fixed Point Exception Register	Record bit (Rc)	Condition Register Field 0
sf	0	CA	0	None
sf.	0	CA	1	LT,GT,EQ,SO
sfo	1	SO,OV,CA	0	None
sfo.	1	SO,OV,CA	1	LT,GT,EQ,SO

The four syntax forms of the **sf** instruction always affect the Carry bit (CA) in the Fixed Point Exception Register. If the syntax form sets the Overflow Exception (OE) bit to 1, the instruction affects the Summary Overflow (SO) and Overflow (OV) bits in the Fixed Point Exception Register. If the syntax form sets the Record (Rc) bit to 1, the instruction affects the Less Than (LT) zero, Greater Than (GT) zero, Equal To (EQ) zero, and Summary Overflow (SO) bits in Condition Register Field 0.

Parameters

RT Specifies target general purpose register where result of operation is stored.

RA Specifies source general purpose register for operation.

RB Specifies source general purpose register for operation.

Examples

1. To subtract the contents of GPR 4 from the contents of GPR 10, store the result in GPR 6, and set the Carry bit to reflect the result of the operation:

```
# Assume GPR 4 contains 0x8000 7000.
# Assume GPR 10 contains 0x9000 3000.
sf 6,4,10
# GPR 6 now contains 0x0FFF C000.
```

2. To subtract the contents of GPR 4 from the contents of GPR 10, store the result in GPR 6, and set Condition Register Field 0 and the Carry bit to reflect the result of the operation:

```
# Assume GPR 4 contains 0x0000 4500.
# Assume GPR 10 contains 0x8000 7000.
sf. 6,4,10
# GPR 6 now contains 0x8000 2B00.
```

3. To subtract the contents of GPR 4 from the contents of GPR 10, store the result in GPR 6, and set the Summary Overflow, Overflow, and Carry bits in the Fixed Point Exception Register to reflect the result of the operation:

```
# Assume GPR 4 contains 0x8000 0000.
# Assume GPR 10 contains 0x0000 4500.
sfo 6,4,10
# GPR 6 now contains 0x8000 4500.
```

4. To subtract the contents of GPR 4 from the contents of GPR 10, store the result in GPR 6, and set the Summary Overflow, Overflow, and Carry bits in the Fixed Point Exception Register and Condition Register Field 0 to reflect the result of the operation:

```
# Assume GPR 4 contains 0x8000 0000.
# Assume GPR 10 contains 0x0000 7000.
sfo. 6,4,10
# GPR 6 now contains 0x8000 7000.
```

Implementation Specifics

This instruction is part of Application Development Toolkit in AIX Base.

Related Information

Understanding Fixed Point Arithmetic Instructions on page 1–8.

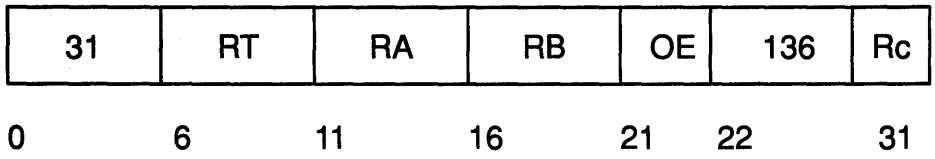
sfe (Subtract From Extended) Instruction

Purpose

Adds the one's complement of the contents of a general purpose register to the sum of a general purpose register and the value of the Fixed Point Exception Register Carry bit and stores the result in a general purpose register.

Syntax

sfe *RT,RA,RB*
sfe. *RT,RA,RB*
sfeo *RT,RA,RB*
sfeo. *RT,RA,RB*



Description

The **sfe** adds the value of the Fixed Point Exception Register Carry bit, the contents of General Purpose Register *RB*, and the one's complement of the contents of General Purpose Register *RA* and stores the result in the target General Purpose Register *RT*.

The **sfe** instruction has four syntax forms. Each syntax form has a different effect on Condition Register Field 0 and the Fixed Point Exception Register.

Syntax form	Overflow Exception (OE)	Fixed Point Exception Register	Record bit (Rc)	Condition Register Field 0
sfe	0	CA	0	None
sfe.	0	CA	1	LT,GT,EQ,SO
sfeo	1	SO,OV,CA	0	None
sfeo.	1	SO,OV,CA	1	LT,GT,EQ,SO

The four syntax forms of the **sfe** instruction always affect the Carry bit (CA) in the Fixed Point Exception Register. If the syntax form sets the Overflow Exception (OE) bit to 1, the instruction affects the Summary Overflow (SO) and Overflow (OV) bits in the Fixed Point Exception Register. If the syntax form sets the Record (Rc) bit to 1, the instruction affects the Less Than (LT) zero, Greater Than (GT) zero, Equal To (EQ) zero, and Summary Overflow (SO) bits in Condition Register Field 0.

Parameters

RT Specifies target general purpose register where result of operation is stored.

RA Specifies source general purpose register for operation.

RB Specifies source general purpose register for operation.

Examples

1. To add the one's complement of the contents of GPR 4, the contents of GPR 10, and the value of the Fixed Point Exception Register Carry bit and store the result in GPR 6:

```
# Assume GPR 4 contains 0x9000 3000.
# Assume GPR 10 contains 0x8000 7000.
# Assume the Carry bit is one.
sfe 6,4,10
# GPR 6 now contains 0xF000 4000.
```

2. To add the one's complement of the contents of GPR 4, the contents of GPR 10, and the value of the Fixed Point Exception Register Carry bit, store the result in GPR 6, and set Condition Register field 0 to reflect the result of the operation:

```
# Assume GPR 4 contains 0x0000 4500.
# Assume GPR 10 contains 0x8000 7000.
# Assume the Carry bit is zero.
sfe. 6,4,10
# GPR 6 now contains 0x8000 2AFF.
```

3. To add the one's complement of the contents of GPR 4, the contents of GPR 10, and the value of the Fixed Point Exception Register Carry bit, store the result in GPR 6, and set the Summary Overflow, Overflow, and Carry bits in the Fixed Point Exception Register to reflect the result of the operation:

```
# Assume GPR 4 contains 0x8000 0000.
# Assume GPR 10 contains 0xEFFF FFFF.
# Assume the Carry bit is one.
sfeo 6,4,10
# GPR 6 now contains 0x6FFF FFFF.
```

4. To add the one's complement of the contents of GPR 4, the contents of GPR 10, and the value of the Fixed Point Exception Register Carry bit, store the result in GPR 6, and set the Summary Overflow, Overflow, and Carry bits in the Fixed Point Exception Register and Condition Register Field 0 to reflect the result of the operation:

```
# Assume GPR 4 contains 0x8000 0000.
# Assume GPR 10 contains 0xEFFF FFFF.
# Assume the Carry bit is zero.
sfeo. 6,4,10
# GPR 6 now contains 0x6FFF FFFE.
```

Implementation Specifics

This instruction is part of Application Development Toolkit in AIX Base.

Related Information

Understanding Fixed Point Arithmetic Instructions on page 1–8.

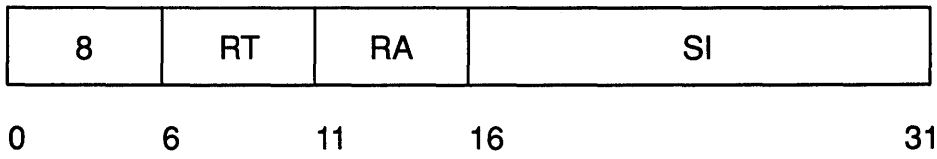
sfi (Subtract From Immediate) Instruction

Purpose

Subtracts the contents of a general purpose register from a 16-bit signed integer and places the result in a general purpose register.

Syntax

sfi *RT,RA,SI*



Description

The **sfi** instruction adds the one's complement of the contents of General Purpose Register *RA*, 1, and a 16-bit signed integer *SI* and places the result in the target General Purpose Register *RT*.

- When *SI* is –1, this instruction places the one's complement of the contents of General purpose Register *RA* in General Purpose Register *RT*.

The **sfi** instruction has one syntax form and does not affect Condition Register Field 0. This instruction always affects the Carry bit in the Fixed Point Exception Register.

Parameters

RT	Specifies target general purpose register where result of operation is stored.
RA	Specifies source general purpose register for operation.
SI	Specifies 16–bit signed integer for operation.

Examples

1. To subtract the contents of GPR 4 from the signed integer 0x0000 7000 and store the result in GPR 6:

```
# Assume GPR 4 holds 0x9000 3000.
sfi 6,4,0x00007000
# GPR 6 now holds 0x7000 4000.
```

Implementation Specifics

This instruction is part of Application Development Toolkit in AIX Base.

Related Information

Understanding Fixed Point Arithmetic Instructions on page 1–8.

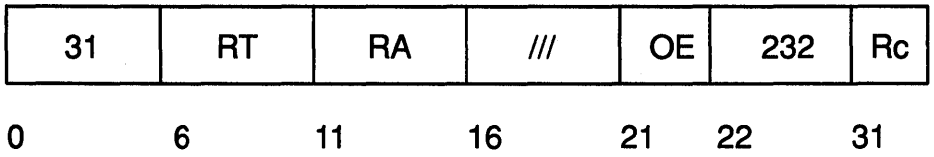
sfme (Subtract From Minus One Extended) Instruction

Purpose

Adds the one's complement of a general purpose register to –1 with carry.

Syntax

sfme *RT,RA*
sfme. *RT,RA*
sfmeo *RT,RA*
sfmeo. *RT,RA*



Description

The **sfme** instruction adds the one's complement of the contents of General Purpose Register *RA*, the Carry Bit of the Fixed Point Exception Register, and x'FFFFFFFF' and places the result in the target General Purpose Register *RT*.

The **sfme** instruction has four syntax forms. Each syntax form has a different effect on Condition Register Field 0 and the Fixed Point Exception Register.

Syntax form	Overflow Exception (OE)	Fixed Point Exception Register	Record bit (Rc)	Condition Register Field 0
sfme	0	CA	0	None
sfme.	0	CA	1	LT,GT,EQ,SO
sfmeo	1	SO,OV,CA	0	None
sfmeo.	1	SO,OV,CA	1	LT,GT,EQ,SO

The four syntax forms of the **sfme** instruction always affect the Carry bit (CA) in the Fixed Point Exception Register. If the syntax form sets the Overflow Exception (OE) bit to 1, the instruction effects the Summary Overflow (SO) and Overflow (OV) bits in the Fixed Point Exception Register. If the syntax form sets the Record (Rc) bit to 1, the instruction affects the Less Than (LT) zero, Greater Than (GT) zero, Equal To (EQ) zero, and Summary Overflow (SO) bits in Condition Register Field 0.

Parameters

RT Specifies target general purpose register where result of operation is stored.

RA Specifies source general purpose register for operation.

Examples

1. To add the one's complement of the contents of GPR 4, the Carry bit of the Fixed Point Excpetion Register, and x'FFFFFFFF' and store the result in PGR 6:

```
# Assume GPR 4 contains 0x9000 3000.
# Assume the Carry bit is set to one.
sfme 6,4
# GPR 6 now contains 0x6FFF CFFF.
```

2. To add the one's complement of the contents of GPR 4, the Carry bit of the Fixed Point Excpetion Register, and x'FFFFFFFF', store the result in GPR 6, and set Condition Register Field 0 to reflect the result of the operation:

```
# Assume GPR 4 contains 0xB004 3000.
# Assume the Carry bit is set to zero.
sfme. 6,4
# GPR 6 now contains 0x4FFB CFFE.
```

3. To add the one's complement of the contents of GPR 4, the Carry bit of the Fixed Point Excpetion Register, and x'FFFFFFFF', store the result in GPR 6, and set the Fixed Point Exception Register to reflect the result of the operation:

```
# Assume GPR 4 contains 0xEFFF FFFF.
# Assume the Carry bit is set to one.
sfmeo 6,4
# GPR 6 now contains 0x1000 0000.
```

4. To add the one's complement of the contents of GPR 4, the Carry bit of the Fixed Point Excpetion Register, and x'FFFFFFFF', store the result in GPR 6, and set Condition Register Field 0 and the Fixed Point Exception Register to reflect the result of the operation:

```
# Assume GPR 4 contains 0xEFFF FFFF.
# Assume the Carry bit is set to zero.
sfmeo. 6,4
# GPR 6 now contains 0x0FFF FFFF.
```

Implementation Specifics

This instruction is part of Application Development Toolkit in AIX Base.

Related Information

Understanding Fixed Point Arithmetic Instructions on page 1–8.

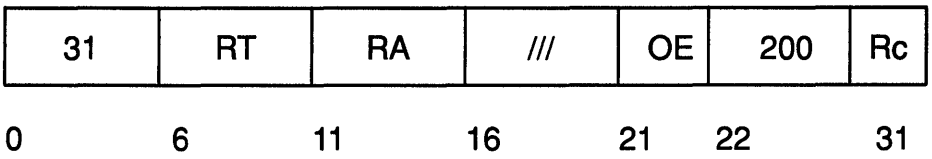
sfze (Subtract From Zero Extended) Instruction

Purpose

Adds the one's complement of the contents of a general purpose register, the Carry bit in the Fixed Point Exception Register, and zero and places the result in a second general purpose register.

Syntax

sfze *RT,RA*
sfze. *RT,RA*
sfzeo *RT,RA*
sfzeo. *RT,RA*



Description

The **sfze** instruction adds the one's complement of the contents of General Purpose Register *RA*, the Carry bit of the Fixed Point Exception Register, and x'00000000' and stores the result in the target General Purpose Register *RT*.

The **sfze** instruction has four syntax forms. Each syntax form has a different effect on Condition Register Field 0 and the Fixed Point Exception Register.

Syntax form	Overflow Exception (OE)	Fixed Point Exception Register	Record bit (Rc)	Condition Register Field 0
sfze	0	CA	0	None
sfze.	0	CA	1	LT,GT,EQ,SO
sfzeo	1	SO,OV,CA	0	None
sfzeo.	1	SO,OV,CA	1	LT,GT,EQ,SO

The four syntax forms of the **sfze** instruction always affect the Carry bit (CA) in the Fixed Point Exception Register. If the syntax form sets the Overflow Exception (OE) bit to 1, the instruction effects the Summary Overflow (SO) and Overflow (OV) bits in the Fixed Point Exception Register. If the syntax form sets the Record (Rc) bit to 1, the instruction affects the Less Than (LT) zero, Greater Than (GT) zero, Equal To (EQ) zero, and Summary Overflow (SO) bits in Condition Register Field 0.

Parameters

RT Specifies target general purpose register where result of operation is stored.

RA Specifies source general purpose register for operation.

Examples

1. To add the one's complement of the contents of GPR 4, the Carry bit, and zero and store the result in GPR 6:

```
# Assume GPR 4 contains 0x9000 3000.
# Assume the Carry bit is set to one.
sfze 6,4
# GPR 6 now contains 0x6FFF D000.
```

2. To add the one's complement of the contents of GPR 4, the Carry bit, and zero, store the result in GPR 6, and set Condition Register Field 0 to reflect the result of the operation:

```
# Assume GPR 4 contains 0xB004 3000.
# Assume the Carry bit is set to one.
sfze. 6,4
# GPR 6 now contains 0x4FFB D000.
```

3. To add the one's complement of the contents of GPR 4, the Carry bit, and zero, store the result in GPR 6, and set the Fixed Point Exception Register to reflect the result of the operation:

```
# Assume GPR 4 contains 0xEFFF FFFF.
# Assume the Carry bit is set to zero.
sfzeo 6,4
# GPR 6 now contains 0x1000 0000.
```

4. To add the one's complement of the contents of GPR 4, the Carry bit, and zero, store the result in GPR 6, and set Condition Register Field 0 and the Fixed Point Exception Register to reflect the result of the operation:

```
# Assume GPR 4 contains 0x70FB 6500.
# Assume the Carry bit is set to zero.
sfzeo 6,4
# GPR 6 now contains 0x8F04 9AFF.
```

Implementation Specifics

This instruction is part of Application Development Toolkit in AIX Base.

Related Information

Understanding Fixed Point Arithmetic Instructions on page 1–8.

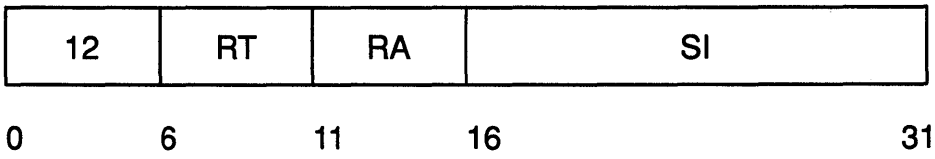
si (Subtract Immediate) Instruction

Purpose

Subtracts the value of a signed integer from the contents of a general purpose register and places the result in a general purpose register.

Syntax

si *RT,RA,SINT*



Description

The **si** instruction subtracts the 16-bit signed integer *SINT* from the contents of General Purpose Register *RA* and stores the result into the target General Purpose Register *RT*. This instruction has the same effect as the **ai** instruction used with a negative *SINT*. The assembler negates *SINT* and places this value (*SI*) in the machine instruction.

```
ai RT,RA,-SINT
```

The **si** instruction has one syntax form and can set the Carry Bit of the Fixed Point Exception Register; it never affects Condition Register Field 0.

Parameters

RT	Specifies target general purpose register for operation.
RA	Specifies specifies source general purpose register for operation.
SINT	Specifies 16–bit signed integer for operation.
SI	Specifies negative of *SINT*.

Examples

1. To subtract 0xFFFF F800 from the contents of GPR 4, store the result in GPR 6, and set the Carry bit in the Fixed Point Exception Register to reflect the result of the operation:

```
# Assume GPR 4 contains 0x0000 0000
si 6,4,0xFFFFF800
# GPR 6 now contains 0x0000 0800
# This instruction has the same effect as
#     ai 6,4,-0xFFFFF800.
```

Implementation Specifics

This instruction is part of Application Development Toolkit in AIX Base.

Related Information

The **ai** (Add Immediate) instruction.

Understanding Fixed Point Arithmetic Instructions on page 1–8.

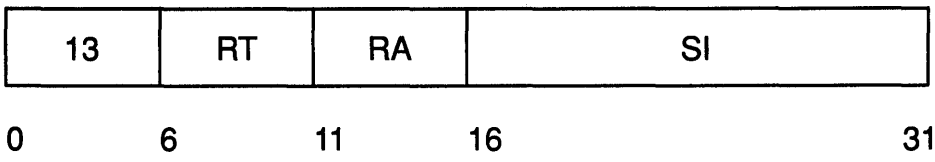
si. (Subtract Immediate and Record) Instruction

Purpose

Subtracts the value of a signed integer from the contents of a general purpose register and places the result in a second general purpose register.

Syntax

si. *RT,RA,SINT*



Description

The **si.** instruction subtracts the 16-bit signed integer *SINT* from the contents of General Purpose Register *RA* and stores the result into the target General Purpose Register *RT*. This instruction has the same effect as the **ai.** instruction used with a negative *SINT*. The assembler negates *SINT* and places this value (*SI*) in the machine instruction.

```
ai. RT,RA,-SINT
```

The **si.** instruction has one syntax form and can set the Carry Bit of the Fixed Point Exception Register. This instruction also affects the Less Than (LT) zero, Greater Than (GT) zero, Equal To (EQ) zero, or Summary Overflow (SO) bit in Condition Register Field 0.

Parameters

RT	Specifies target general purpose register for operation.
RA	Specifies specifies source general purpose register for operation.
SINT	Specifies 16–bit signed integer for operation.
SI	Specifies negative of *SINT*.

Examples

1. To subtract 0xFFFF F800 from the contents of GPR 4, store the result in GPR 6, and set the Carry bit in the Fixed Point Exception Register and Condition Register Field 0 to reflect the result of the operation:

```
# Assume GPR 4 contains 0xEFFF FFFF.
si. 6,4,0xFFFFF800
# GPR 6 now contains 0xF000 07FF.
# This instruction has the same effect as
#     ai. 6,4,-0xFFFFF800.
```

Implementation Specifics

This instruction is part of Application Development Toolkit in AIX Base.

Related Information

The **ai.** (Add Immediate and Record) instruction.

Understanding Fixed Point Arithmetic Instructions on page 1–8.

sl (Shift Left) Instruction

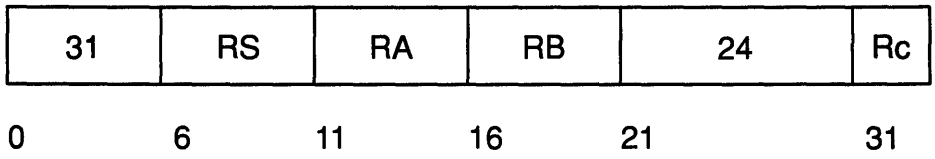
Purpose

Rotates the contents of a general purpose register left by a number of bits and places the masked result in another general purpose register.

Syntax

sl *RA,RS,RB*

sl. *RA,RS,RB*



Description

The **sl** instruction rotates the contents of the source General Purpose Register *RS* left *N* bits, where *N* is the shift amount specified in bits 27–31 of General Purpose Register *RB*, and stores the logical AND of the rotated word and the generated mask in General Purpose Register *RA*.

- If bit 26 of register *RB* is zero, then a mask of 32–*N* ones followed by *N* zeros is generated.
- If bit 26 of register *RB* is one, then a mask of all zeros is generated.

The **sl** instruction has two syntax forms. Each syntax form has a different effect on Condition Register Field 0.

Syntax form	Overflow Exception (OE)	Fixed Point Exception Register	Record bit (Rc)	Condition Register Field 0
sl	None	None	0	None
sl.	None	None	1	LT,GT,EQ,SO

The two syntax forms of the **sl** instruction never affect the Fixed Point Exception Register. If the syntax form sets the Record (Rc) bit to 1, the instruction affects the Less Than (LT) zero, Greater Than (GT) zero, Equal To (EQ) zero, and Summary Overflow (SO) bits in Condition Register Field 0.

Parameters

RA Specifies target general purpose register where result of operation is stored.

RS Specifies source general purpose register for operation.

RB Specifies source general purpose register for operation.

Examples

1. To rotate the contents of GPR 4 to the left 15 bits and store the result of ANDing the rotated data with a generated mask in GPR 6:

```
# Assume GPR 5 contains 0x0000 002F.
# Assume GPR 4 contains 0xFFFF FFFF.
sl 6,4,5
# GPR 6 now contains 0x0000 0000.
```

2. To rotate the contents of GPR 4 to the left 5 bits, store the result of ANDing the rotated data with a generated mask in GPR 6, and set Condition Register Field 0 to reflect the result of the operation:

```
# Assume GPR 4 contains 0xB004 3000.
# Assume GPR 5 contains 0x0000 0005.
sl. 6,4,5
# GPR 6 now contains 0x0086 0000.
# Condition Register Field 0 now contains 0x4.
```

Implementation Specifics

This instruction is part of Application Development Toolkit in AIX Base.

Related Information

Understanding Fixed Point Shift Instructions on page 1–10.

sle (Shift Left Extended) Instruction

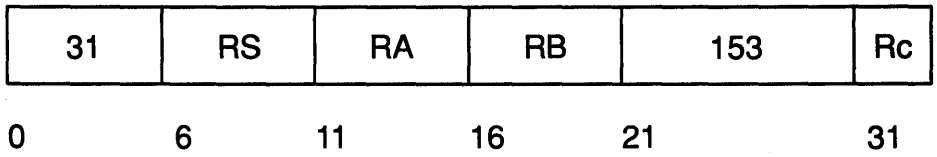
Purpose

Shifts the contents of a general purpose register left by a number of bits and places a copy of the rotated data in the MQ register and the result in a general purpose register.

Syntax

sle *RA,RS,RB*

sle. *RA,RS,RB*



Description

The **sle** instruction rotates the contents of the source General Purpose Register *RS* left *N* bits, where *N* is the shift amount specified in bits 27–31 of General Purpose Register *RB*, and stores the rotated word in the MQ register and the logical AND of the rotated word and the generated mask in General Purpose Register *RA*. The mask consists of 32 minus *N* ones followed by *N* zeros.

The **sle** instruction has two syntax forms. Each syntax form has a different effect on Condition Register Field 0.

Syntax form	Overflow Exception (OE)	Fixed Point Exception Register	Record bit (Rc)	Condition Register Field 0
sl	None	None	0	None
sl.	None	None	1	LT,GT,EQ,SO

The two syntax forms of the **sle** instruction never affect the Fixed Point Exception Register. If the syntax form sets the Record (Rc) bit to 1, the instruction affects the Less Than (LT) zero, Greater Than (GT) zero, Equal To (EQ) zero, and Summary Overflow (SO) bits in Condition Register Field 0.

Parameters

RA Specifies target general purpose register where result of operation is stored.

RS Specifies source general purpose register for operation.

RB Specifies source general purpose register for operation.

Examples

1. To rotate the contents of GPR 4 to the left 4 bits, place a copy of the rotated data in the MQ Register, and place the result of ANDing the rotated data with a mask into GPR 6:

```
# Assume GPR 4 contains 0x9000 3000.
# Assume GPR 5 contains 0x0000 0004.
sle 6,4,5
# GPR 6 now contains 0x0003 0000.
# The MQ Register now contains 0x0003 0009.
```

2. To rotate the contents of GPR 4 to the left 4 bits, place a copy of the rotated data in the MQ Register, place the result of ANDing the rotated data with a mask into GPR 6, and set Condition Register Field 0 to reflect the result of the operation:

```
# Assume GPR 4 contains 0xB004 3000.
# Assume GPR 5 contains 0x0000 0004.
sle. 6,4,5
# GPR 6 now contains 0x0043 0000.
# The MQ Register now contains 0x0043 000B.
# Condition Register Field 0 now contains 0x4.
```

Implementation Specifics

This instruction is part of Application Development Toolkit in AIX Base.

Related Information

Understanding Fixed Point Shift Instructions on page 1–10.

sleq (Shift Left Extended with MQ) Instruction

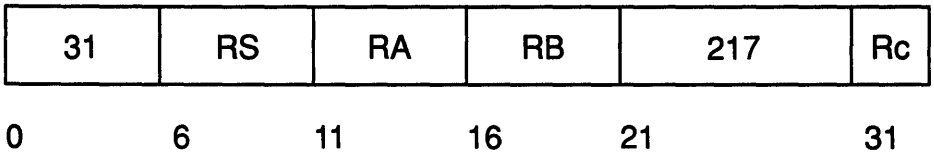
Purpose

Rotates the contents of a general purpose register left by a number of bits, merges the result with the contents of the MQ register under control of a mask, and places the rotated word in the MQ Register and the masked result in a general purpose register.

Syntax

sleq *RA,RS,RB*

sleq. *RA,RS,RB*



Description

The **sleq** instruction rotates the contents of the source General Purpose Register *RS* left *N* bits, where *N* is the shift amount specified in bits 27–31 of General Purpose Register *RB*, merges the rotated word with the contents of the MQ register under control of a mask, and stores the rotated word in the MQ Register and merged word in General Purpose Register *RA*. The mask consists of 32 minus *N* ones followed by *N* zeros.

The **sleq** instruction has two syntax forms. Each syntax form has a different effect on Condition Register Field 0.

Syntax form	Overflow Exception (OE)	Fixed Point Exception Register	Record bit (Rc)	Condition Register Field 0
sleq	None	None	0	None
sleq.	None	None	1	LT,GT,EQ,SO

The two syntax forms of the **sleq** instruction never affect the Fixed Point Exception Register. If the syntax form sets the Record (Rc) bit to 1, the instruction affects the Less Than (LT) zero, Greater Than (GT) zero, Equal To (EQ) zero, and Summary Overflow (SO) bits in Condition Register Field 0.

Parameters

RA Specifies target general purpose register where result of operation is stored.

RS Specifies source general purpose register for operation.

RB Specifies source general purpose register for operation.

Examples

1. To rotate the contents of GPR 4 to the left 4 bits, merge the rotated data with the contents of the MQ register under a generated mask, and place the rotated word in the MQ register and the result in GPR 6 :

```
# Assume GPR 4 contains 0x9000 3000.
# Assume GPR 5 contains 0x0000 0004.
# Assume the MQ Register contains 0xFFFF FFFF.
sleq 6,4,5
# GPR 6 now contains 0x0003 000F.
# The MQ register now contains 0x0003 0009.
```

2. To rotate the contents of GPR 4 to the left 4 bits, merge the rotated data with the contents of the MQ register under a generated mask, place the rotated word in the MQ register and the result in GPR 6, and set Condition Register Field 0 to reflect the result of the operation:

```
# Assume GPR 4 contains 0xB004 3000.
# Assume GPR 5 contains 0x0000 0004.
# Assume the MQ Register contains 0xFFFF FFFF.
sleq. 6,4,5
# GPR 6 now contains 0x0043 000F.
# The MQ register now contains 0x0043 000B.
# Condition Register Field 0 now contains 0x4.
```

Implementation Specifics

This instruction is part of Application Development Toolkit in AIX Base.

Related Information

Understanding Fixed Point Shift Instructions on page 1–10.

sliq (Shift Left Immediate with MQ) Instruction

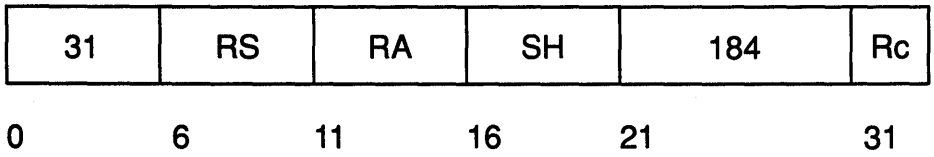
Purpose

Shifts the contents of a general purpose register left by a number of bits in an immediate value, and places the rotated contents in the MQ Register and the result in a general purpose register.

Syntax

sliq *RA,RS,SH*

sliq. *RA,RS,SH*



Description

The **sliq** instruction rotates the contents of the source General Purpose Register *RS* left *N* bits, where *N* is the shift amount specified by *SH*, and stores the rotated word in the MQ Register and the logical AND of the rotated word and the generated mask in General Purpose Register *RA*. The mask consists of 32 minus *N* ones followed by *N* zeros.

The **sliq** instruction has two syntax forms. Each syntax form has a different effect on Condition Register Field 0.

Syntax form	Overflow Exception (OE)	Fixed Point Exception Register	Record bit (Rc)	Condition Register Field 0
sliq	None	None	0	None
sliq.	None	None	1	LT,GT,EQ,SO

The two syntax forms of the **sliq** instruction never affect the Fixed Point Exception Register. If the syntax form sets the Record (Rc) bit to 1, the instruction affects the Less Than (LT) zero, Greater Than (GT) zero, Equal To (EQ) zero, and Summary Overflow (SO) bits in Condition Register Field 0.

Parameters

RA Specifies target general purpose register where result of operation is stored.

RS Specifies source general purpose register for operation.

SH Specifies immediate value for shift amount.

Examples

1. To rotate the contents of GPR 4 to the left 20 bits, AND the rotated data with a generated mask, and place the rotated word into the MQ Register and the result in GPR 6:

```
# Assume GPR 4 contains 0x1234 5678.
sliq 6,4,0x14
# GPR 6 now contains 0x6780 0000.
# MQ Register now contains 0x6781 2345.
```

2. To rotate the contents of GPR 4 to the left 16 bits, AND the rotated data with a generated mask, place the rotated word into the MQ Register and the result in GPR 6, and set Condition Register Field 0 to reflect the result of the operation:

```
# Assume GPR 4 contains 0x1234 5678.
sliq. 6,4,0x10
# GPR 6 now contains 0x5678 0000.
# The MQ Register now contains 0x5678 1234.
# Condition Register Field 0 now contains 0x4.
```

Implementation Specifics

This instruction is part of Application Development Toolkit in AIX Base.

Related Information

Understanding Fixed Point Shift Instructions on page 1–10.

slliq (Shift Left Long Immediate With MQ) Instruction

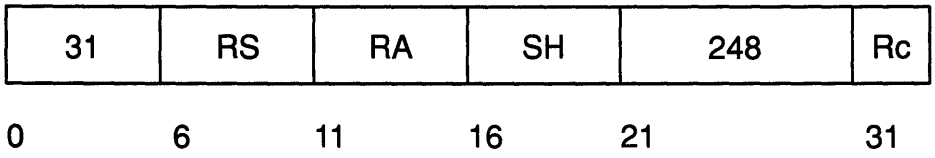
Purpose

Rotates the contents of a general purpose register left by a number of bits in an immediate value, merges the result with the contents of the MQ Register under control of a mask, and places the rotated word in the MQ Register and the masked result in another general purpose register.

Syntax

slliq *RA,RS,SH*

slliq. *RA,RS,SH*



Description

The **slliq** instruction rotates the contents of the source General Purpose Register *RS* left *N* bits, where *N* is the shift amount specified in *SH*, merges the result with the contents of the MQ register, and stores the rotated word in the MQ register and the final result in General Purpose Register *RA*. The mask consists of 32 minus *N* ones followed by *N* zeros.

The **slliq** instruction has two syntax forms. Each syntax form has a different effect on Condition Register Field 0.

Syntax form	Overflow Exception (OE)	Fixed Point Exception Register	Record bit (Rc)	Condition Register Field 0
slliq	None	None	0	None
slliq.	None	None	1	LT,GT,EQ,SO

The two syntax forms of the **slliq** instruction never affect the Fixed Point Exception Register. If the syntax form sets the Record (Rc) bit to 1, the instruction affects the Less Than (LT) zero, Greater Than (GT) zero, Equal To (EQ) zero, and Summary Overflow (SO) bits in Condition Register Field 0.

Parameters

RA Specifies target general purpose register where result of operation is stored.

RS Specifies source general purpose register for operation.

SH Specifies immediate value for shift amount.

Examples

1. To rotate the contents of GPR 4 to the left 3 bits, merge the rotated data with the contents of the MQ register under a generated mask, and place the rotated word in the MQ Register and the result in GPR 6:

```
# Assume GPR 4 contains 0x9000 3000.
# Assume the MQ Register contains 0xFFFF FFFF.
slliq 6,4,0x3
# GPR 6 now contains 0x8001 8007.
# The MQ Register now contains 0x8001 8004.
```

2. To rotate the contents of GPR 4 to the left 4 bits, merge the rotated data with the contents of the MQ register under a generated mask, place the rotated word in the MQ Register and the result in GPR 6, and set Condition Register Field 0 to reflect the result of the operation:

```
# Assume GPR 4 contains 0xB004 3000.
# Assume the MQ Register contains 0xFFFF FFFF.
slliq. 6,4,0x4
# GPR 6 now contains 0x0043 000F.
# The MQ Register contains 0x0043 000B.
# Condition Register Field 0 now contains 0x4.
```

Implementation Specifics

This instruction is part of Application Development Toolkit in AIX Base.

Related Information

Understanding Fixed Point Shift Instructions on page 1–10.

sllq (Shift Left Long with MQ) Instruction

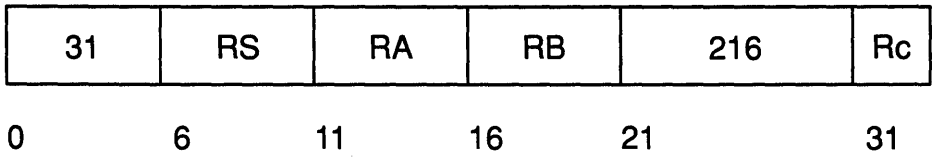
Purpose

Rotates the contents of a general purpose register left by a number of bits specified in a general purpose register, merges either the rotated data or a word of zeros with the contents of the MQ Register, and places the result in general purpose register.

Syntax

sllq *RA,RS,RB*

sllq. *RA,RS,RB*



Description

The **sllq** instruction rotates the contents of the source General Purpose Register *RS* left *N* bits, where *N* is the shift amount specified in bits 27–31 of General Purpose Register *RB*. The merge depends on the value of bit 26 in General Purpose Register *RB*.

- If bit 26 of General Purpose Register *RB* is zero, then a mask of *N* zeros followed by 32 minus *N* ones is generated. The rotated word is then merged with the contents of the MQ Register under control of this generated mask
- If bit 26 of General Purpose Register *RB* is one, then a mask of *N* ones followed by 32 minus *N* zeros is generated. A word of zeros is then merged with the contents of the MQ Register under control of this generated mask

The resulting merged word is stored in General Purpose Register *RA*. The MQ Register is not altered.

The **sllq** instruction has two syntax forms. Each syntax form has a different effect on Condition Register Field 0.

Syntax form	Overflow Exception (OE)	Fixed Point Exception Register	Record bit (Rc)	Condition Register Field 0
sllq	None	None	0	None
sllq.	None	None	1	LT,GT,EQ,SO

The two syntax forms of the **sllq** instruction never affect the Fixed Point Exception Register. If the syntax form sets the Record (Rc) bit to 1, the instruction affects the Less Than (LT) zero, Greater Than (GT) zero, Equal To (EQ) zero, and Summary Overflow (SO) bits in Condition Register Field 0.

Parameters

RA Specifies target general purpose register where result of operation is stored.

RS Specifies source general purpose register for operation.

RB Specifies source general purpose register for operation.

Examples

1. To rotate the contents of GPR 4 to the left 4 bits, merge a word of zeros with the contents of the MQ register under a mask, and place the merged result in GPR 6:

```
# Assume GPR 4 contains 0x9000 3000.
# Assume GPR 5 contains 0x0000 0024.
# Assume MQ Register contains 0xABCD EFAB.
sllq 6,4,5
# GPR 6 now contains 0xABCD EFA0.
# The MQ Register remains unchanged.
```

2. To rotate the contents of GPR 4 to the left 4 bits, merge the rotated data with the contents of the MQ register under a mask, place the merged result in GPR 6, and set Condition Register Field 0 to reflect the result of the operation:

```
# Assume GPR 4 contains 0xB004 3000.
# Assume GPR 5 contains 0x0000 0004.
# Assume MQ Register contains 0xFFFF FFFF.
sllq. 6,4,5
# GPR 6 now contains 0x0043 000F.
# The MQ Register remains unchanged.
# Condition Register Field 0 now contains 0x4.
```

Implementation Specifics

This instruction is part of Application Development Toolkit in AIX Base.

Related Information

Understanding Fixed Point Shift Instructions on page 1–10.

slq (Shift Left with MQ) Instruction

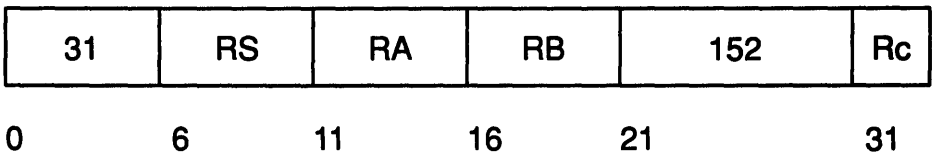
Purpose

Rotates the contents of a general purpose register left by a number of bits specified in a general purpose register, places the rotated word in the MQ Register, and places the logical AND of the rotated word and a generated mask in a general purpose register.

Syntax

slq *RA,RS,RB*

slq. *RA,RS,RB*



Description

The **slq** instruction rotates the contents of the source General Purpose Register *RS* left *N* bits, where *N* is the shift amount specified in bits 27–31 of General Purpose Register *RB*, and stores the rotated word in the MQ Register. The mask depends on bit 26 of General Purpose Register *RB*.

- If bit 26 of General Purpose Register *RB* is zero, then a mask of 32 minus *N* ones followed by *N* zeros is generated.
- If bit 26 of General Purpose Register *RB* is one, then a mask of all zeros is generated.

This instruction then stores the logical AND of the rotated word and the generated mask in General Purpose Register *RA*.

The **slq** instruction has two syntax forms. Each syntax form has a different effect on Condition Register Field 0.

Syntax form	Overflow Exception (OE)	Fixed Point Exception Register	Record bit (Rc)	Condition Register Field 0
slq	None	None	0	None
slq.	None	None	1	LT,GT,EQ,SO

The two syntax forms of the **slq** instruction never affect the Fixed Point Exception Register. If the syntax form sets the Record (Rc) bit to 1, the instruction affects the Less Than (LT) zero, Greater Than (GT) zero, Equal To (EQ) zero, and Summary Overflow (SO) bits in Condition Register Field 0.

Parameters

RA Specifies target general purpose register where result of operation is stored.

RS Specifies source general purpose register for operation.

RB Specifies source general purpose register for operation.

Examples

1. To rotate the contents of GPR 4 to the left 4 bits, place the rotated word in the MQ Register, and place logical AND of the rotated word and the generated mask in GPR 6:

```
# Assume GPR 4 contains 0x9000 3000.
# Assume GPR 5 contains 0x0000 0024.
slq 6,4,5
# GPR 6 now contains 0x0000 0000.
# The MQ Register now contains 0x0003 0009.
```

2. To rotate the contents of GPR 4 to the left 4 bits, place the rotated word in the MQ Register, place logical AND of the rotated word and the generated mask in GPR 6, and set Condition Register Field 0 to reflect the result of the operation:

```
# Assume GPR 4 contains 0xB004 3000.
# Assume GPR 5 contains 0x0000 0004.
slq. 6,4,5
# GPR 6 now contains 0x0043 0000.
# The MQ Register now contains 0x0043 000B.
# Condition Register Field 0 now contains 0x4.
```

Implementation Specifics

This instruction is part of Application Development Toolkit in AIX Base.

Related Information

Understanding Fixed Point Shift Instructions on page 1–10.

sr (Shift Right) Instruction

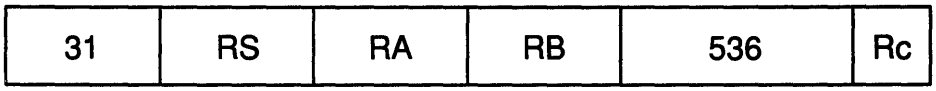
Purpose

Rotates the contents of a general purpose register left by a number of bits and places the masked result in a general purpose register.

Syntax

sr *RA,RS,RB*

sr. *RA,RS,RB*



Description

The **sr** instruction rotates the contents of the source General Purpose Register *RS* left 32 minus *N* bits, where *N* is the shift amount specified in bits 27–31 of General Purpose Register *RB*, and stores the logical AND of the rotated word and the generated mask in General Purpose Register *RA*.

- If bit 26 of register *RB* is zero, then a mask of *N* zeros followed by 32–*N* ones is generated.
- If bit 26 of register *RB* is one, then a mask of all zeros is generated.

The **sr** instruction has two syntax forms. Each syntax form has a different effect on Condition Register Field 0.

Syntax form	Overflow Exception (OE)	Fixed Point Exception Register	Record bit (Rc)	Condition Register Field 0
sr	None	None	0	None
sr.	None	None	1	LT,GT,EQ,SO

The two syntax forms of the **sr** instruction never affect the Fixed Point Exception Register. If the syntax form sets the Record (Rc) bit to 1, the instruction affects the Less Than (LT) zero, Greater Than (GT) zero, Equal To (EQ) zero, and Summary Overflow (SO) bits in Condition Register Field 0.

Parameters

RA Specifies target general purpose register where result of operation is stored.

RS Specifies source general purpose register for operation.

RB Specifies source general purpose register for operation.

Examples

1. To rotate the contents of GPR 4 to the left 28 bits and store the result of ANDing the rotated data with a generated mask in GPR 6:

```
# Assume GPR 4 contains 0x9000 3000.
# Assume GPR 5 contains 0x0000 0024.
sr 6,4,5
# GPR 6 now contains 0x0000 0000.
```

2. To rotate the contents of GPR 4 to the left 28 bits, store the result of ANDing the rotated data with a generated mask in GPR 6, and set Condition Register Field 0 to reflect the result of the operation:

```
# Assume GPR 4 contains 0xB004 3001.
# Assume GPR 5 contains 0x0000 0004.
sr. 6,4,5
# GPR 6 now contains 0x0B00 4300.
# Condition Register Field 0 now contains 0x4.
```

Implementation Specifics

This instruction is part of Application Development Toolkit in AIX Base.

Related Information

Understanding Fixed Point Shift Instructions on page 1–10.

sra (Shift Right Algebraic) Instruction

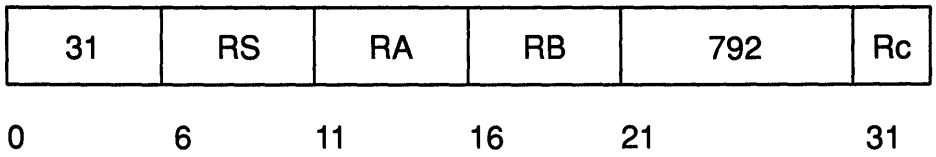
Purpose

Rotates the contents of a general purpose register left by a number of bits, merges the rotated data with a word of 32 sign bits from that register under control of a generated mask, and places the result in a general purpose register.

Syntax

sra *RA,RS,RB*

sra. *RA,RS,RB*



Description

The **sra** instruction rotates the contents of the source General Purpose Register *RS* left 32 minus *N* bits, where *N* is the shift amount specified in bits 27–31 of General Purpose Register *RB*, and merges the rotated word with a word of 32 sign bits from General Purpose Register *RS* under control of a generated mask. A word of 32 sign bits is generated by taking the sign bit of a general purpose register and repeating it 32 times to make a full word. This word can be either 0x0000 0000 or 0xFFFF FFFF depending on the value of the general purpose register.

The mask depends on the value of bit 26 in General Purpose Register *RB*.

- If bit 26 of General Purpose Register *RB* is zero, then a mask of *N* zeros followed by 32 minus *N* ones is generated.
- If bit 26 of General Purpose Register *RB* is one, then a mask of all zeros is generated.

The merged word is placed in General Purpose Register *RA*.This instruction then ANDs the rotated data with the complement of the generated mask, ORs the 32–bit result together, and ANDs the bit result with bit 0 of General Purpose Register *RS* to produce the Carry bit (CA).

The **sra** instruction has two syntax forms. Each syntax form has a different effect on Condition Register Field 0.

Syntax form	Overflow Exception (OE)	Fixed Point Exception Register	Record bit (Rc)	Condition Register Field 0
sra	None	CA	0	None
sra.	None	CA	1	LT,GT,EQ,SO

The two syntax forms of the **sra** instruction always affect the Carry bit (CA) in the Fixed Point Exception Register. If the syntax form sets the Record (Rc) bit to 1, the instruction effects the Less Than (LT) zero, Greater Than (GT) zero, Equal To (EQ) zero, and Summary Overflow (SO) bits in Condition Register Field 0.

Parameters

RA	Specifies target general purpose register where result of operation is stored.
RS	Specifies source general purpose register for operation.
RB	Specifies source general purpose register for operation.

Examples

1. To rotate the contents of GPR 4 to the left 28 bits, merge the result with 32 sign bits under control of a generated mask, store the result in GPR 6, and set the Carry bit in the Fixed Point Exception Register to reflect the result of the operation:

```
# Assume GPR 4 contains 0x9000 3000.
# Assume GPR 5 contains 0x0000 0024.
sra 6,4,5
# GPR 6 now contains 0xFFFF FFFF.
```

2. To rotate the contents of GPR 4 to the left 28 bits, merge the result with 32 sign bits under control of a generated mask, store the result in GPR 6, and set the Carry bit in the Fixed Point Exception Register and Condition Register Field 0 to reflect the result of the operation:

```
# Assume GPR 4 contains 0xB004 3000.
# Assume GPR 5 contains 0x0000 0004.
sra. 6,4,5
# GPR 6 now contains 0xFB00 4300.
# Condition Register Field 0 now contains 0x8.
```

Implementation Specifics

This instruction is part of Application Development Toolkit in AIX Base.

Related Information

The **aze** (Add To Zero Extended) instruction.

Understanding Fixed Point Shift Instructions on page 1–10.

srai (Shift Right Algebraic Immediate) Instruction

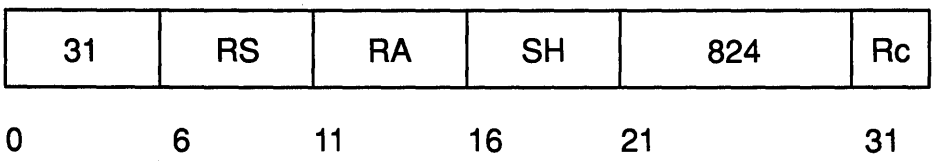
Purpose

Rotates the contents of a general purpose register a specified number of bits to the left, merges the rotated data with a word of 32 sign bits from that register under control of a generated mask, and places the result in a general purpose register.

Syntax

srai *RA,RS,SH*

srai. *RA,RS,SH*



Description

The **srai** instruction rotates the contents of the source General Purpose Register *RS* left 32 minus *N* bits, where *N* is the shift amount specified by *SH*, merges the rotated data with a word of 32 sign bits from General Purpose Register *RS* under control of a generated mask, and stores the merged result in General Purpose Register *RA*. A word of 32 sign bits is generated by taking the sign bit of a general purpose register and repeating it 32 times to make a full word. This word can be either 0x0000 0000 or 0xFFFF FFFF depending on the value of the general purpose register. The mask consists of *N* zeros followed by 32 minus *N* ones.

This instruction then ANDs the rotated data with the complement of the generated mask, ORs the 32–bit result together, and ANDs the bit result with bit 0 of General Purpose Register *RS* to produce the Carry bit (CA).

The **srai** instruction has two syntax forms. Each syntax form has a different effect on Condition Register Field 0.

Syntax form	Overflow Exception (OE)	Fixed Point Exception Register	Record bit (Rc)	Condition Register Field 0
srai	None	CA	0	None
srai.	None	CA	1	LT,GT,EQ,SO

The two syntax forms of the **srai** instruction always affect the Carry bit (CA) in the Fixed Point Exception Register. If the syntax form sets the Record (Rc) bit to 1, the instruction affects the Less Than (LT) zero, Greater Than (GT) zero, Equal To (EQ) zero, and Summary Overflow (SO) bits in Condition Register Field 0.

Parameters

RA Specifies target general purpose register where result of operation is stored.

RS Specifies source general purpose register for operation.

SH Specifies immediate value for shift amount.

Examples

1. To rotate the contents of GPR 4 to the left 28 bits, merge the result with 32 sign bits under control of a generated mask, store the result in GPR 6, and set the Carry bit in the Fixed Point Exception Register to reflect the result of the operation:

```
# Assume GPR 4 contains 0x9000 3000.
srai 6,4,0x4
# GPR 6 now contains 0xF900 0300.
```

2. To rotate the contents of GPR 4 to the left 28 bits, merge the result with 32 sign bits under control of a generated mask, place the result in GPR 6, and set the Carry bit in the Fixed Point Exception Register and Condition Register Field 0 to reflect the result of the operation:

```
# Assume GPR 4 contains 0xB004 3000.
srai. 6,4,0x4
# GPR 6 now contains 0xFB00 4300.
# Condition Register Field 0 now contains 0x8.
```

Implementation Specifics

This instruction is part of Application Development Toolkit in AIX Base.

Related Information

The **aze** (Add To Zero Extended) instruction.

Understanding Fixed Point Shift Instructions on page 1–10.

sraiq (Shift Right Algebraic Immediate With MQ) Instruction

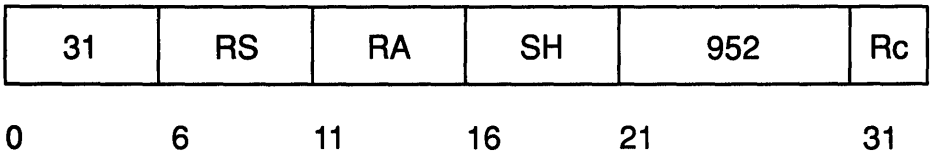
Purpose

Rotates the contents of a general purpose register left by a specified number of bits, merges the rotated data with a word of 32 sign bits from that general purpose register under control of a generated mask, and places the rotated word in the MQ Register and the merged result in a general purpose register.

Syntax

sraiq *RA,RS,SH*

sraiq. *RA,RS,SH*



Description

The **sraiq** instruction rotates the contents of the source General Purpose Register *RS* left 32 minus *N* bits, where *N* is the shift amount specified by *SH*, merges the rotated data with a word of 32 sign bits from General Purpose Register *RS* under control of a generated mask, and stores the rotated word in the MQ Register and the merged result in General Purpose Register *RA*. A word of 32 sign bits is generated by taking the sign bit of a general purpose register and repeating it 32 times to make a full word. This word can be either 0x0000 0000 or 0xFFFF FFFF depending on the value of the general purpose register. The mask consists of *N* zeros followed by 32 minus *N* ones.

This instruction then ANDs the rotated data with the complement of the generated mask, ORs the 32–bit result together, and ANDs the bit result with bit 0 of General Purpose Register *RS* to produce the Carry bit (CA).

The **sraiq** instruction has two syntax forms. Each syntax form has a different effect on Condition Register Field 0.

Syntax form	Overflow Exception (OE)	Fixed Point Exception Register	Record bit (Rc)	Condition Register Field 0
sraiq	None	CA	0	None
sraiq.	None	CA	1	LT,GT,EQ,SO

The two syntax forms of the **sraiq** instruction always affect the Carry bit (CA) in the Fixed Point Exception Register. If the syntax form sets the Record (Rc) bit to 1, the instruction affects the Less Than (LT) zero, Greater Than (GT) zero, Equal To (EQ) zero, and Summary Overflow (SO) bits in Condition Register Field 0.

Parameters

RA Specifies target general purpose register where result of operation is stored.

RS Specifies source general purpose register for operation.

SH Specifies immediate value for shift amount.

Examples

1. To rotate the contents of GPR 4 to the left 28 bits, merge the result with 32 sign bits under control of a generated mask, store the result in GPR 6, and set the Carry bit in the Fixed Point Exception Register to reflect the result of the operation:

```
# Assume GPR 4 contains 0x9000 3000.
sraiq 6,4,0x4
# GPR 6 now contains 0xF900 0300.
# MQ now contains 0x0900 0300.
```

2. To rotate the contents of GPR 4 to the left 28 bits, merge the result with 32 sign bits under control of a generated mask, store the result in GPR 6, and set the Carry bit in the Fixed Point Exception Register and Condition Register Field 0 to reflect the result of the operation:

```
# Assume GPR 4 contains 0xB004 3000.
sraiq. 6,4,0x4
# GPR 6 now contains 0xFB00 4300.
# MQ now contains 0x0B00 4300.
# Condition Register Field 0 now contains 0x8.
```

Implementation Specifics

This instruction is part of Application Development Toolkit in AIX Base.

Related Information

The **aze** (Add To Zero Extended) instruction.

Understanding Fixed Point Shift Instructions on page 1–10.

sraq (Shift Right Algebraic With MQ) Instruction

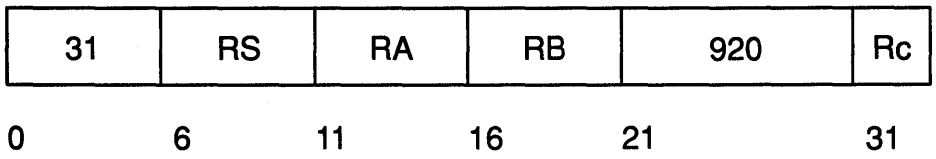
Purpose

Rotates a general purpose register a specified number of bits to the left, merges the result with a word of 32 sign bits from that general purpose register under control of a generated mask, and places the rotated word in the MQ Register and the merged result in a general purpose register.

Syntax

sraq *RA,RS,RB*

sraq. *RA,RS,RB*



Description

The **sraq** instruction rotates the contents of the source General Purpose Register *RS* left 32 minus *N* bits, where *N* is the shift amount specified in bits 27–31 of General Purpose Register *RB*. The instruction then merges the rotated data with a word of 32 sign bits from General Purpose Register *RS* under control of a generated mask and stores the merged word in General Purpose Register *RA*. The rotated word is stored in the MQ Register. The mask depends on the value of bit 26 in General Purpose Register *RB*.

- If bit 26 of General Purpose Register *RB* is zero, then a mask of *N* zeros followed by 32 minus *N* ones is generated.
- If bit 26 of General Purpose Register *RB* is one, then a mask of all zeros is generated.

A word of 32 sign bits is generated by taking the sign bit of a general purpose register and repeating it 32 times to make a full word. This word can be either 0x0000 0000 or 0xFFFF FFFF depending on the value of the general purpose register.

This instruction then ANDs the rotated data with the complement of the generated mask, ORs the 32–bit result together, and ANDs the bit result with bit 0 of General Purpose Register *RS* to produce the Carry bit (CA).

The **sraq** instruction has two syntax forms. Each syntax form has a different effect on Condition Register Field 0.

Syntax form	Overflow Exception (OE)	Fixed Point Exception Register	Record bit (Rc)	Condition Register Field 0
sraq	None	CA	0	None
sraq.	None	CA	1	LT,GT,EQ,SO

The two syntax forms of the **sraq** instruction always affect the Carry bit (CA) in the Fixed Point Exception Register. If the syntax form sets the Record (Rc) bit to 1, the instruction effects the Less Than (LT) zero, Greater Than (GT) zero, Equal To (EQ) zero, and Summary Overflow (SO) bits in Condition Register Field 0.

Parameters

RA	Specifies target general purpose register where result of operation is stored.
RS	Specifies source general purpose register for operation.
RB	Specifies source general purpose register for operation.

Examples

1. To rotate the contents of GPR 4 to the left 28 bits, merge the result with 32 sign bits under control of a generated mask, place the result in GPR 6 and the rotated word in the MQ Register, and set the Carry bit in the Fixed Point Exception Register to reflect the result of the operation:

```
# Assume GPR 4 contains 0x9000 3000.
# Assume GPR 7 contains 0x0000 0024.
sraq 6,4,7
# GPR 6 now contains 0xFFFF FFFF.
# The MQ Register now contains 0x0900 0300.
```

2. To rotate the contents of GPR 4 to the left 28 bits, merge the result with 32 sign bits under control of a generated mask, place the result in GPR 6 and the rotated word in the MQ Register, and set the Carry bit in the Fixed Point Exception Register and Condition Register Field 0 to reflect the result of the operation:

```
# Assume GPR 4 contains 0xB004 3000.
# Assume GPR 7 contains 0x0000 0004.
sraq. 6,4,7
# GPR 6 now contains 0xFB00 4300.
# The MQ Register now contains 0x0B00 4300.
# Condition Register Field 0 now contains 0x4.
```

Implementation Specifics

This instruction is part of Application Development Toolkit in AIX Base.

Related Information

The **aze** (Add To Zero Extended) instruction.

Understanding Fixed Point Shift Instructions on page 1–10.

sre (Shift Right Extended) Instruction

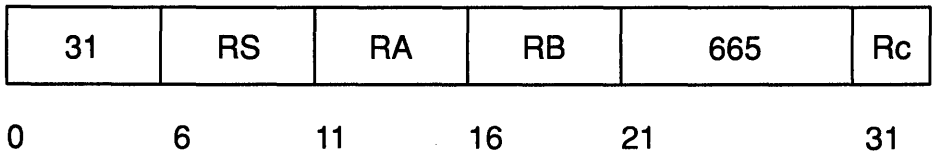
Purpose

Shifts the contents of a general purpose register right by a number of bits and places a copy of the rotated data in the MQ register and the result in a general purpose register.

Syntax

sre *RA,RS,RB*

sre. *RA,RS,RB*



Description

The **sre** instruction rotates the contents of the source General Purpose Register *RS* left 32 minus *N* bits, where *N* is the shift amount specified in bits 27–31 of General Purpose Register *RB*, and stores the rotated word in the MQ register and the logical AND of the rotated word and a generated mask in General Purpose Register *RA*. The mask consists of *N* zeros followed by 32 minus *N* ones.

The **sre** instruction has two syntax forms. Each syntax form has a different effect on Condition Register Field 0.

Syntax form	Overflow Exception (OE)	Fixed Point Exception Register	Record bit (Rc)	Condition Register Field 0
sre	None	None	0	None
sre.	None	None	1	LT,GT,EQ,SO

The two syntax forms of the **sre** instruction never affect the Fixed Point Exception Register. If the syntax form sets the Record (Rc) bit to 1, the instruction affects the Less Than (LT) zero, Greater Than (GT) zero, Equal To (EQ) zero, and Summary Overflow (SO) bits in Condition Register Field 0.

Parameters

RA Specifies target general purpose register where result of operation is stored.

RS Specifies source general purpose register for operation.

RB Specifies source general purpose register for operation.

Examples

1. To rotate the contents of GPR 4 to the left 20 bits, place a copy of the rotated data in the MQ Register, and place the the result of ANDing the rotated data with a mask into GPR 6:

```
# Assume GPR 4 contains 0x9000 3000.
# Assume GPR 5 contains 0x0000 000C.
sre 6,4,5
# GPR 6 now contains 0x0009 0003.
# The MQ Register now contains 0x0009 0003.
```

2. To rotate the contents of GPR 4 to the left 17 bits, place a copy of the rotated data in the MQ Register, place the the result of ANDing the rotated data with a mask into GPR 6, and set Condition Register Field 0 to reflect the result of the operation:

```
# Assume GPR 4 contains 0xB004 3000.
# Assume GPR 5 contains 0x0000 000F.
sre. 6,4,5
# GPR 6 now contains 0x0001 6008.
# The MQ Register now contains 0x6001 6008.
# Condition Register Field 0 now contains 0x4.
```

Implementation Specifics

This instruction is part of Application Development Toolkit in AIX Base.

Related Information

Understanding Fixed Point Shift Instructions on page 1–10.

srea (Shift Right Extended Algebraic) Instruction

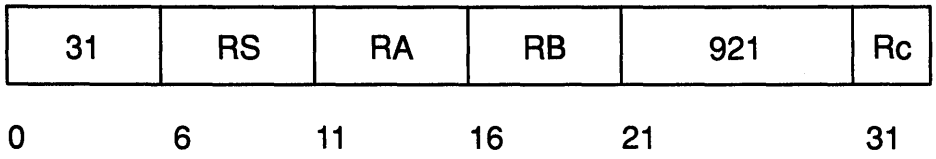
Purpose

Rotates the contents of a general purpose register left by a number of bits, places a copy of the rotated data in the MQ Register, merges the rotated word and a word of 32 sign bits from the general purpose register under control of a mask, and places the result in a general purpose register.

Syntax

srea *RA,RS,RB*

srea. *RA,RS,RB*



Description

The **sre** instruction rotates the contents of the source General Purpose Register *RS* left 32 minus *N* bits, where *N* is the shift amount specified in bits 27–31 of General Purpose Register *RB*, stores the rotated word in the MQ Register, and merges the rotated word and a word of 32 sign bits from General Purpose Register *RS* under control of a generated mask. A word of 32 sign bits is generated by taking the sign bit of a general purpose register and repeating it 32 times to make a full word. This word can be either 0x0000 0000 or 0xFFFF FFFF depending on the value of the general purpose register. The mask consists of *N* zeros followed by 32 minus *N* ones. The merged word is stored in General Purpose Register *RA*.

This instruction then ANDs the rotated data with the complement of the generated mask, ORs together the 32–bit result, and ANDs the bit result with bit 0 of General Purpose Register *RS* to produce the Carry bit (CA).

The **srea** instruction has two syntax forms. Each syntax form has a different effect on Condition Register Field 0.

Syntax form	Overflow Exception (OE)	Fixed Point Exception Register	Record bit (Rc)	Condition Register Field 0
srea	None	CA	0	None
srea	None	CA	1	LT,GT,EQ,SO

The two syntax forms of the **srea** instruction always affect the Carry bit (CA) in the Fixed Point Exception Register. If the syntax form sets the Record (Rc) bit to 1, the instruction affects the Less Than (LT) zero, Greater Than (GT) zero, Equal To (EQ) zero, and Summary Overflow (SO) bits in Condition Register Field 0.

Parameters

RA Specifies target general purpose register where result of operation is stored.

RS Specifies source general purpose register for operation.

RB Specifies source general purpose register for operation.

Examples

1. To rotate the contents of GPR 4 to the left 28 bits, merge the result with 32 sign bits under control of a generated mask, place the rotated word in the MQ Register and the result in GPR 6, and set the Carry bit in the Fixed Point Exception Register to reflect the result of the operation:

```
# Assume GPR 4 contains 0x9000 3000.
# Assume GPR 7 contains 0x0000 0004.
srea 6,4,7
# GPR 6 now contains 0xF900 0300.
# The MQ Register now contains 0x0900 0300.
```

2. To rotate the contents of GPR 4 to the left 28 bits, merge the result with 32 sign bits under control of a generated mask, place the rotated word in the MQ Register and the result in GPR 6, and set the Carry bit in the Fixed Point Exception Register and Condition Register Field 0 to reflect the result of the operation:

```
# Assume GPR 4 contains 0xB004 3000.
# Assume GPR 7 contains 0x0000 0004.
srea. 6,4,7
# GPR 6 now contains 0xFB00 4300.
# The MQ Register now contains 0x0B00 4300.
# Condition Register Field 0 now contains 0x8.
```

Implementation Specifics

This instruction is part of Application Development Toolkit in AIX Base.

Related Information

The **aze** (Add To Zero Extended) instruction.

Understanding Fixed Point Shift Instructions on page 1–10.

sreq (Shift Right Extended With MQ) Instruction

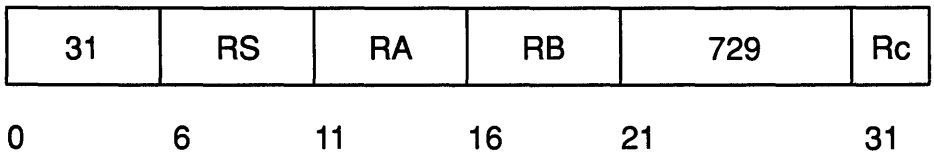
Purpose

Rotates the contents of a general purpose register left by a number of bits, merges the result with the contents of the MQ Register under control of a generated mask, and places the rotated word in the MQ Register and the merged result in a general purpose register.

Syntax

sreq *RA,RS,RB*

sreq. *RA,RS,RB*



Description

The **sreq** instruction rotates the contents of the source General Purpose Register *RS* left 32 minus *N* bits, where *N* is the shift amount specified in bits 27–31 of General Purpose Register *RB*, merges the rotated word with the contents of the MQ register under a generated mask, and stores the rotated word in the MQ Register and the merged word in General Purpose Register *RA*. The mask consists of *N* zeros followed by 32 minus *N* ones.

The **sreq** instruction has two syntax forms. Each syntax form has a different effect on Condition Register Field 0.

Syntax form	Overflow Exception (OE)	Fixed Point Exception Register	Record bit (Rc)	Condition Register Field 0
sreq	None	None	0	None
sreq.	None	None	1	LT,GT,EQ,SO

The two syntax forms of the **sreq** instruction never affect the Fixed Point Exception Register. If the syntax form sets the Record (Rc) bit to 1, the instruction affects the Less Than (LT) zero, Greater Than (GT) zero, Equal To (EQ) zero, and Summary Overflow (SO) bits in Condition Register Field 0.

Parameters

RA Specifies target general purpose register where result of operation is stored.

RS Specifies source general purpose register for operation.

RB Specifies source general purpose register for operation.

Examples

1. To rotate the contents of GPR 4 to the left 28 bits, merge the rotated data with the contents of the MQ register under a generated mask, and place the rotated word in the MQ register and the result in GPR 6 :

```
# Assume GPR 4 contains 0x9000 300F.
# Assume GPR 7 contains 0x0000 0004.
# Assume the MQ Register contains 0xEFFF FFFF.
sreq 6,4,7
# GPR 6 now contains 0xE900 0300.
# The MQ Register now contains 0xF900 0300.
```

2. To rotate the contents of GPR 4 to the left 28 bits, merge the rotated data with the contents of the MQ register under a generated mask, place the rotated word in the MQ register and the result in GPR 6, and set Condition Register Field 0 to reflect the result of the operation:

```
# Assume GPR 4 contains 0xB00 300F.
# Assume GPR 18 contains 0x0000 0004.
# Assume the MQ Register contains 0xEFFF FFFF
sreq. 6,4,18
# GPR 6 now contains 0xEB00 0300.
# The MQ Register now contains 0xFB00 0300.
# Condition Register Field 0 now contains 0x8.
```

Implementation Specifics

This instruction is part of Application Development Toolkit in AIX Base.

Related Information

Understanding Fixed Point Shift Instructions on page 1–10.

sriq (Shift Right Immediate With MQ) Instruction

Purpose

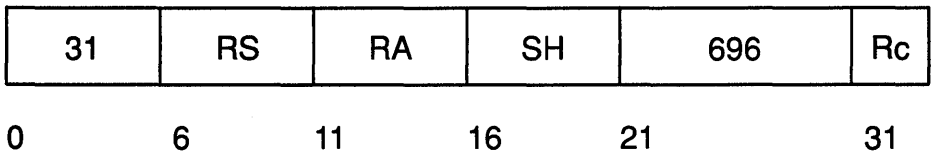
Shifts the contents of a general purpose register right by a number of bits and places the rotated contents in the MQ Register and the result in a general purpose register.

Syntax

sriq *RA,RS,SH*

sriq. *RA,RS,SH*

Extended mnemonics are also provided.



Description

The **sriq** instruction rotates the contents of the source General Purpose Register *RS* left 32 minus *N* bits, where *N* is the shift amount specified by *SH*, and stores the rotated word in the MQ Register and the logical AND of the rotated word and the generated mask in General Purpose Register *RA*. The mask consists of *N* zeros followed by 32 minus *N* ones.

The **sriq** instruction has two syntax forms. Each syntax form has a different effect on Condition Register Field 0.

Syntax form	Overflow Exception (OE)	Fixed Point Exception Register	Record bit (Rc)	Condition Register Field 0
sriq	None	None	0	None
sriq.	None	None	1	LT,GT,EQ,SO

The two syntax forms of the **sriq** instruction never affect the Fixed Point Exception Register. If the syntax form sets the Record (Rc) bit to 1, the instruction affects the Less Than (LT) zero, Greater Than (GT) zero, Equal To (EQ) zero, and Summary Overflow (SO) bits in Condition Register Field 0.

Parameters

RA Specifies target general purpose register where result of operation is stored.

RS Specifies source general purpose register for operation.

SH Specifies value for shift amount.

Examples

1. To rotate the contents of GPR 4 to the left 20 bits, AND the rotated data with a generated mask, and place the rotated word into the MQ Register and the result in GPR 6:

```
# Assume GPR 4 contains 0x9000 300F.
sriq 6,4,0xC
# GPR 6 now contains 0x0009 0003.
# The MQ Register now contains 0x00F9 0003.
```

2. To rotate the contents of GPR 4 to the left 12 bits, AND the rotated data with a generated mask, place the rotated word into the MQ Register and the result in GPR 6, and set Condition Register Field 0 to reflect the result of the operation:

```
# Assume GPR 4 contains 0xB000 300F.
sriq. 6,4,0x14
# GPR 6 now contains 0x0000 0B00.
# The MQ Register now contains 0x0300 FB00.
# Condition Register Field 0 now contains 0x4.
```

Implementation Specifics

This instruction is part of Application Development Toolkit in AIX Base.

Related Information

Understanding Fixed Point Shift Instructions on page 1–10.

srliq (Shift Right Long Immediate With MQ) Instruction

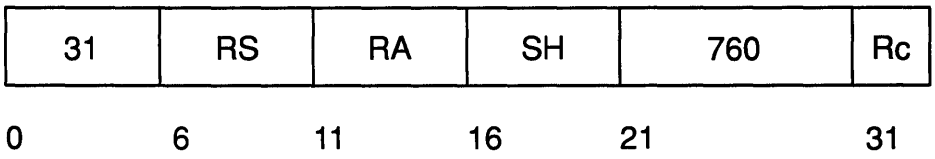
Purpose

Rotates the contents of a general purpose register left by a number of bits, merges the result with the contents of the MQ Register under control of a generated mask, and places the result in another general purpose register.

Syntax

srliq *RA,RS,SH*

srliq. *RA,RS,SH*



Description

The **srliq** instruction rotates the contents of the source General Purpose Register *RS* left 32 minus *N* bits, where *N* is the shift amount specified by *SH*, merges the result with the contents of the MQ Register under control of a generated mask, and stores the rotated word in the MQ Register and the merged result in General Purpose Register *RA*. The mask consists of *N* zeros followed by 32 minus *N* ones.

The **srliq** instruction has two syntax forms. Each syntax form has a different effect on Condition Register Field 0.

Syntax form	Overflow Exception (OE)	Fixed Point Exception Register	Record bit (Rc)	Condition Register Field 0
srliq	None	None	0	None
srliq.	None	None	1	LT,GT,EQ,SO

The two syntax forms of the **srliq** instruction never affect the Fixed Point Exception Register. If the syntax form sets the Record (Rc) bit to 1, the instruction affects the Less Than (LT) zero, Greater Than (GT) zero, Equal To (EQ) zero, and Summary Overflow (SO) bits in Condition Register Field 0.

Parameters

RA Specifies target general purpose register where result of operation is stored.

RS Specifies source general purpose register for operation.

SH Specifies value for shift amount.

Examples

1. To rotate the contents of GPR 4 to the left 28 bits, merge the rotated data with the contents of the MQ Register under a generated mask, and place the rotated word in the MQ Register and the result in GPR 6:

```
# Assume GPR 4 contains 0x9000 300F.
# Assume the MQ Register contains 0x1111 1111.
srliq 6,4,0x4
# GPR 6 now contains 0x1900 0300.
# The MQ Register now contains 0xF900 0300.
```

2. To rotate the contents of GPR 4 to the left 28 bits, merge the rotated data with the contents of the MQ Register under a generated mask, place the rotated word in the MQ Register and the result in GPR 6, and set Condition Register Field 0 to reflect the result of the operation:

```
# Assume GPR 4 contains 0xB004 3000
# Assume the MQ Register contains 0xFFFF FFFF.
srliq. 6,4,0x4
# GPR 6 now contains 0xFB00 4300.
# The MQ Register contains 0x0B00 4300.
# Condition Register Field 0 now contains 0x8.
```

Implementation Specifics

This instruction is part of Application Development Toolkit in AIX Base.

Related Information

Understanding Fixed Point Shift Instructions on page 1–10.

srlq (Shift Right Long With MQ) Instruction

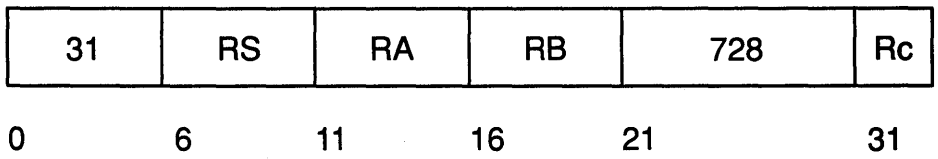
Purpose

Rotates the contents of a general purpose register left by a number of bits, merges either the rotated data or a word of zeros with the contents of the MQ Register under control of a generated mask, and places the result in a general purpose register.

Syntax

srlq *RA,RS,RB*

srlq. *RA,RS,RB*



Description

The **srlq** instruction rotates the contents of the source General Purpose Register *RS* left 32 minus *N* bits, where *N* is the shift amount specified in bits 27–31 of General Purpose Register *RB*. The merge depends on the value of bit 26 in General Purpose Register *RB*.

- If bit 26 of General Purpose Register *RB* is zero, then a mask of *N* zeros followed by 32 minus *N* ones is generated. The rotated word is then merged with the contents of the MQ Register under control of this generated mask
- If bit 26 of General Purpose Register *RB* is one, then a mask of *N* ones followed by 32 minus *N* zeros is generated. A word of zeros is then merged with the contents of the MQ Register under control of this generated mask

The merged word is stored in General Purpose Register *RA*. The MQ Register is not altered.

The **srlq** instruction has two syntax forms. Each syntax form has a different effect on Condition Register Field 0.

Syntax form	Overflow Exception (OE)	Fixed Point Exception Register	Record bit (Rc)	Condition Register Field 0
srlq	None	None	0	None
srlq.	None	None	1	LT,GT,EQ,SO

The two syntax forms of the **srlq** instruction never affect the Fixed Point Exception Register. If the syntax form sets the Record (Rc) bit to 1, the instruction affects the Less Than (LT) zero, Greater Than (GT) zero, Equal To (EQ) zero, and Summary Overflow (SO) bits in Condition Register Field 0.

Parameters

RA Specifies target general purpose register where result of operation is stored.

RS Specifies source general purpose register for operation.

RB Specifies source general purpose register for operation.

Examples

1. To rotate the contents of GPR 4 to the left 28 bits, merge a word of zeros with the contents of the MQ register under a mask, and place the merged result in GPR 6:

```
# Assume GPR 4 contains 0x9000 300F.
# Assume GPR 8 contains 0x0000 0024.
# Assume the MQ Register contains 0xFFFF FFFF.
srlq 6,4,8
# GPR 6 now contains 0x0FFF FFFF.
# The MQ Register remains unchanged.
```

2. To rotate the contents of GPR 4 to the left 28 bits, merge the rotated data with the contents of the MQ register under a mask, place the merged result in GPR 6, and set Condition Register Field 0 to reflect the result of the operation:

```
# Assume GPR 4 contains 0xB004 3000.
# Assume GPR 8 contains 0x00000 0004.
# Assume the MQ Register contains 0xFFFF FFFF.
srlq. 6,4,8
# GPR 6 now holds 0xFB00 4300.
# The MQ Register remains unchanged.
# Condition Register Field 0 now contains 0x8.
```

Implementation Specifics

This instruction is part of Application Development Toolkit in AIX Base.

Related Information

Understanding Fixed Point Shift Instructions on page 1–10.

srq (Shift Right with MQ) Instruction

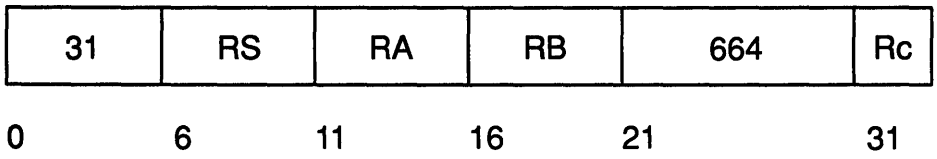
Purpose

Rotates the contents of a general purpose register left by a number of bits, places the rotated word in the MQ Register, and places the logical AND of the rotated word and a generated mask in a general purpose register.

Syntax

srq *RA,RS,RB*

srq. *RA,RS,RB*



Description

The **srq** instruction rotates the contents of the source General Purpose Register *RS* left 32 minus *N* bits, where *N* is the shift amount specified in bits 27–31 of General Purpose Register *RB*, and stores the rotated word in the MQ Register. The mask depends on bit 26 of General Purpose Register *RB*.

- If bit 26 of General Purpose Register *RB* is zero, then a mask of *N* zeros followed by 32 minus *N* ones is generated.
- If bit 26 of General Purpose Register *RB* is one, then a mask of all zeros is generated.

This instruction then stores the logical AND of the rotated word and the generated mask in General Purpose Register *RA*.

The **srq** instruction has two syntax forms. Each syntax form has a different effect on Condition Register Field 0.

Syntax form	Overflow Exception (OE)	Fixed Point Exception Register	Record bit (Rc)	Condition Register Field 0
srq	None	None	0	None
srq.	None	None	1	LT,GT,EQ,SO

The two syntax forms of the **srq** instruction never affect the Fixed Point Exception Register. If the syntax form sets the Record (Rc) bit to 1, the instruction affects the Less Than (LT) zero, Greater Than (GT) zero, Equal To (EQ) zero, and Summary Overflow (SO) bits in Condition Register Field 0.

Parameters

RA Specifies target general purpose register where result of operation is stored.

RS Specifies source general purpose register for operation.

RB Specifies source general purpose register for operation.

Examples

1. To rotate the contents of GPR 4 to the left 28 bits, place the rotated word in the MQ Register, and place logical AND of the rotated word and the generated mask in GPR 6:

```
# Assume GPR 4 holds 0x9000 300F.
# Assume GPR 25 holds 0x0000 00024.
srq 6,4,25
# GPR 6 now holds 0x0000 0000.
# The MQ Register now holds 0xF900 0300.
```

2. To rotate the contents of GPR 4 to the left 28 bits, place the rotated word in the MQ Register, place logical AND of the rotated word and the generated mask in GPR 6, and set Condition Register Field 0 to reflect the result of the operation:

```
# Assume GPR 4 holds 0xB000 300F.
# Assume GPR 25 holds 0x0000 0004.
srq. 6,4,8
# GPR 6 now holds 0x0B00 0300.
# The MQ Register now holds 0xFB00 0300.
# Condition Register Field 0 now contains 0x4.
```

Implementation Specifics

This instruction is part of Application Development Toolkit in AIX Base.

Related Information

Understanding Fixed Point Shift Instructions on page 1–10.

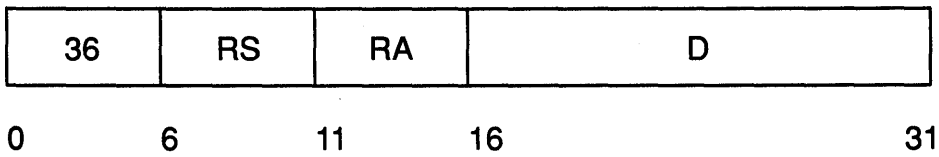
st (Store) Instruction

Purpose

Stores a word of data from a general purpose register into a specified location in memory.

Syntax

st *RS,D(RA)*



Description

The **st** instruction stores a word from General Purpose Register *RS* into a word of storage addressed by the effective address (EA).

The effective address (EA) is the sum of the contents of General Purpose Register *RA* and *D*, a 16-bit signed two's complement integer sign extended to 32 bits, if *RA* is not 0. If *RA* is 0, then the effective address (EA) is *D*.

- If alignment checking is disabled, i.e., the alignment bit (AL) in the Machine Status Register (MSR) is 0, then the two low order bits of the effective address are ignored.
- If alignment checking is enabled, and the two low order bits are not b'00', then the hardware attempts to perform the unaligned storage access. If the hardware cannot perform the unaligned storage access, an Alignment Interrupt is generated.

The **st** instruction has one syntax form and does not affect the Fixed Point Exception Register or Condition Register Field 0.

Parameters

RS	Specifies source general purpose register of stored data.
D	16-bit signed two's complement integer sign extended to 32 bits for EA calculation.
RA	Specifies source general purpose register for EA calculation.

Examples

1. To store the contents of GPR 6 into a location in memory:

```
.csect data[rw]
buffer: .long 0,0
# Assume GPR 6 contains 0x9000 3000.
# Assume GPR 5 contains the address of buffer.
.csect text[pr]
st 6,4(5)
# 0x9000 3000 is now stored at the address buffer+4.
```

Implementation Specifics

This instruction is part of Application Development Toolkit in AIX Base.

Related Information

Understanding Fixed Point Store Instructions on page 1–7.

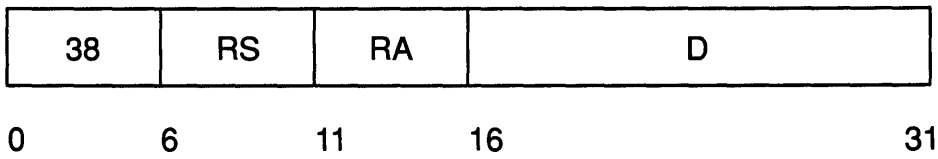
stb (Store Byte) Instruction

Purpose

Stores a byte of data from a general purpose register into a specified location in memory.

Syntax

stb *RS,D(RA)*



Description

The **stb** instruction stores bits 24–31 of General Purpose Register *RS* into a byte of storage addressed by the effective address (EA).

The effective address (EA) is the sum of the contents of General Purpose Register *RA* and *D*, a 16-bit signed two's complement integer sign extended to 32 bits, if *RA* is not 0. If *RA* is 0, then the effective address (EA) is *D*.

The **stb** instruction has one syntax form and does not affect the Fixed Point Exception Register or Condition Register Field 0.

Parameters

RS	Specifies source general purpose register of stored data.
D	16-bit signed two's complement integer sign extended to 32 bits for EA calculation.
RA	Specifies source general purpose register for EA calculation.

Examples

1. To store bits 24–31 of GPR 6 into a location in memory:

```
.csect data[rw]
buffer: .long 0
# Assume GPR 4 contains address of csect data[rw].
# Assume GPR 6 contains 0x0000 0060.
.csect text[pr]
stb 6,buffer(4)
# 0x60 is now stored at the address of buffer.
```

Implementation Specifics

This instruction is part of Application Development Toolkit in AIX Base.

Related Information

Understanding Fixed Point Store Instructions on page 1–7.

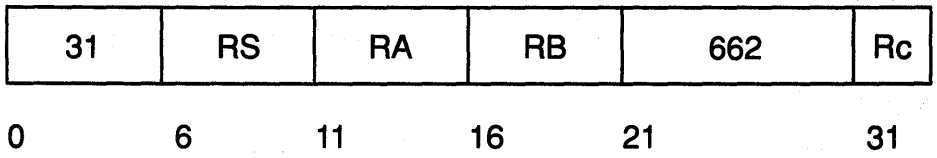
stbrx (Store Byte Reverse Indexed) Instruction

Purpose

Stores a byte–reversed word of data from a general purpose register into a specified location in memory.

Syntax

stbrx *RS,RA,RB*



Description

The **stbrx** instruction stores a byte–reversed word from General Purpose Register *RS* into a word of storage addressed by the effective address (EA).

- Bits 24–31 of GPR *RS* are stored into bits 00–07 of the word in storage addressed by EA.
- Bits 16–23 of GPR *RS* are stored into bits 08–15 of the word in storage addressed by EA.
- Bits 08–15 of GPR *RS* are stored into bits 16–23 of the word in storage addressed by EA.
- Bits 00–07 of GPR *RS* are stored into bits 24–31 of the word in storage addressed by EA.

The effective address (EA) is the sum of the contents of General Purpose Register *RA* and General Purpose Register *RB* if *RA* is not 0. If *RA* is 0, then the effective address (EA) is the contents of *RB*.

- If alignment checking is disabled, i.e., the alignment bit (AL) in the Machine Status Register (MSR) is 0, then the two low order bits of the effective address are ignored.
- If alignment checking is enabled, and the two low order bits are not b'00', then the hardware attempts to perform the unaligned storage access. If the hardware cannot perform the unaligned storage access, an Alignment Interrupt is generated.

The **stbrx** instruction has one syntax form and does not affect the Fixed Point Exception Register or Condition Register Field 0.

Parameters

RS	Specifies source general purpose register of stored data.
RA	Specifies source general purpose register for EA calculation.
RB	Specifies source general purpose register for EA calculation.

Examples

1. To store a byte–reverse word from GPR 6 into a location in memory:

```
.csect data[rw]
buffer: .long 0
# Assume GPR 4 contains the address of buffer.
# Assume GPR 9 contains 0x0000 0000.
# Assume GPR 6 contains 0x1234 5678.
.csect text[pr]
stbrx 6,4,9
# 0x7856 3412 is now stored at the address of buffer.
```

Implementation Specifics

This instruction is part of Application Development Toolkit in AIX Base.

Related Information

Understanding Fixed Point Store Instructions on page 1–7.

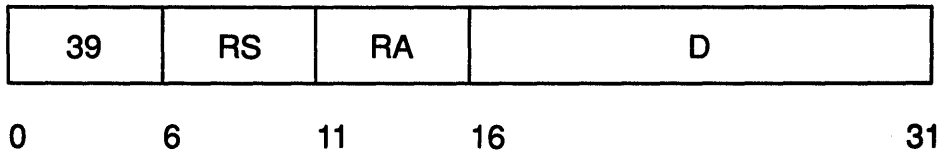
stbu (Store Byte With Update) Instruction

Purpose

Stores a byte of data from a general purpose register into a specified location in memory and possibly places the address in a general purpose register.

Syntax

stbu *RS,D(RA)*



Description

The **stbu** instruction stores bits 24–31 of the source General Purpose Register *RS* into the byte in storage addressed by the effective address (EA).

The effective address (EA) is the sum of the contents of General Purpose Register *RA* and *D*, a 16-bit signed two's complement integer sign extended to 32 bits, if *RA* is not 0. If *RA* is 0, then the effective address (EA) is *D*.

- If *RA* does not equal 0 and the storage access does not cause an Alignment Interrupt, then the effective address is stored in General Purpose Register *RA*.

The **stbu** instruction has one syntax form and does not affect the Fixed Point Exception Register or Condition Register Field 0.

Parameters

RS	Specifies source general purpose register of stored data.
D	16-bit signed two's complement integer sign extended to 32 bits for EA calculation.
RA	Specifies source general purpose register for EA calculation and possible address update.

Examples

1. To store bits 24–31 of GPR 6 into a location in memory and place the address in GPR 16:

```
.csect data[rw]
buffer: .long 0
# Assume GPR 6 contains 0x0000 0060.
# Assume GPR 16 contains the address of csect data[rw].
.csect text[pr]
stbu 6,buffer(16)
# GPR 16 now contains the address of buffer.
# 0x60 is stored at the address of buffer.
```

Implementation Specifics

This instruction is part of Application Development Toolkit in AIX Base.

Related Information

Understanding Fixed Point Store with Update Instructions on page 1–7.

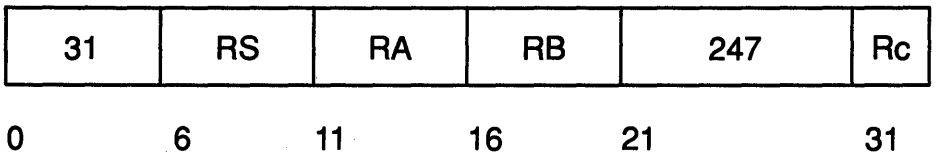
stbux (Store Byte With Update Indexed) Instruction

Purpose

Stores a byte of data from a general purpose register into a specified location in memory and possibly places the address in a general purpose register.

Syntax

stbux *RS,RA,RB*



Description

The **stbux** instruction stores bits 24–31 of the source General Purpose Register *RS* into the byte in storage addressed by the effective address (EA).

The effective address (EA) is the sum of the contents of General Purpose Register *RA* and the contents of General Purpose Register *RB*, if *RA* is not 0. If *RA* is 0, then the effective address (EA) is the contents of *RB*.

- If *RA* does not equal 0 and the storage access does not cause an Alignment Interrupt, then the effective address is stored in General Purpose Register *RA*.

The **stbux** instruction exists only in one syntax form and does not affect the Fixed Point Exception Register or Condition Register Field 0.

Parameters

RS	Specifies source general purpose register of stored data.
RA	Specifies source general purpose register for EA calculation and possible address update.
RB	Specifies source general purpose register for EA calculation.

Examples

1. To store the contents of GPR 6 into a location in memory and place the address in GPR 4:

```
.csect data[rw]
buffer: .long 0
# Assume GPR 6 contains 0x0000 0060.
# Assume GPR 4 contains 0x0000 0000.
# Assume GPR 19 contains the address of buffer.
.csect text[pr]
stbux 6,4,19
# Buffer now contains 0x60.
# GPR 4 contains the address of buffer.
```

Implementation Specifics

This instruction is part of Application Development Toolkit in AIX Base.

Related Information

Understanding Fixed Point Store with Update Instructions on page 1–7.

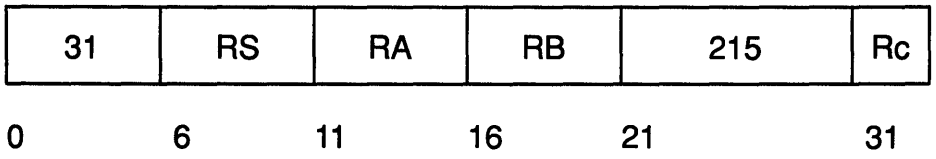
stbx (Store Byte Indexed) Instruction

Purpose

Stores a byte from a general purpose register into a specified location in memory.

Syntax

stbx *RS,RA,RB*



Description

The **stbx** instruction stores bits 24–31 from General Purpose Register *RS* into a byte of storage addressed by the effective address (EA). The contents of General Purpose Register *RS* are unchanged.

The effective address (EA) is the sum of the contents of General Purpose Register *RA* and the contents of General Purpose Register *RB*, if *RA* is not 0. If *RA* is 0, then the effective address (EA) is the contents of *RB*.

The **stbx** instruction has one syntax form and does not affect the Fixed Point Exception Register or Condition Register Field 0.

Parameters

RS	Specifies source general purpose register of stored data.
RA	Specifies source general purpose register for EA calculation.
RB	Specifies source general purpose register for EA calculation.

Examples

1. To store bits 24–31 of GPR 6 into a location in memory:

```
.csect data[rw]
buffer: .long 0
# Assume GPR 4 contains the address of buffer.
# Assume GPR 6 contains 0x4865 6C6F.
.csect text[pr]
stbx 6,0,4
# buffer now contains 0x6F.
```

Implementation Specifics

This instruction is part of Application Development Toolkit in AIX Base.

Related Information

Understanding Fixed Point Store Instructions on page 1–7.

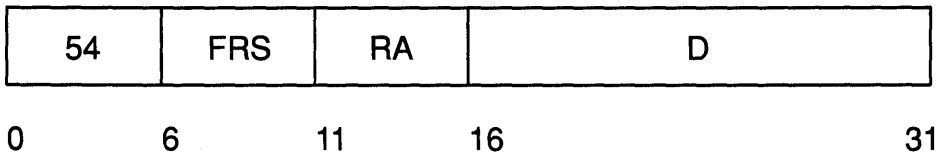
stfd (Store Floating Point Double) Instruction

Purpose

Stores a double word of data in a specified location in memory.

Syntax

stfd *FRS,D(RA)*



Description

The **stfd** instruction stores the contents of Floating Point Register *FRS* into the doubleword storage addressed by the effective address (EA).

The effective address (EA) is the sum of the contents of General Purpose Register *RA* and *D*, a 16-bit signed two's complement integer sign extended to 32 bits, if *RA* is not 0. If *RA* is 0, then the effective address (EA) is *D*.

- If alignment checking is disabled, i.e., the alignment bit (AL) in the Machine Status Register (MSR) is 0, then the three low order bits of the effective address are ignored.
- If alignment checking is enabled, and the three low order bits are not b'000', then the hardware attempts to perform the unaligned storage access. If the hardware cannot perform the unaligned storage access, an Alignment Interrupt is generated.

The **stfd** instruction has one syntax form and does not affect the Fixed Point Exception Register or Condition Register Field 0.

Parameters

FRS	Specifies source floating point register of stored data.
D	16-bit signed two's complement integer sign extended to 32 bits for EA calculation.
RA	Specifies source general purpose register for EA calculation.

Examples

1. To store the contents of FPR 6 into a location in memory:

```
.csect data[rw]
buffer: .long 0,0
# Assume FPR 6 contains 0x4865 6C6C 6F20 776F.
# Assume GPR 4 contains the address of csect data[rw].
.csect text[pr]
stfd 6,buffer(4)
# buffer now contains 0x4865 6C6C 6F20 776F.
```

Implementation Specifics

This instruction is part of Application Development Toolkit in AIX Base.

Related Information

Understanding Floating Point Store Instructions on page 1–13.

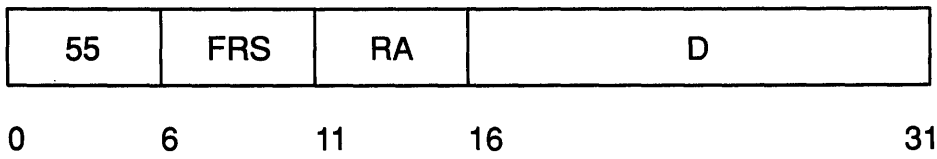
stfdu (Store Floating Point Double With Update) Instruction

Purpose

Stores a double word of data in a specified location in memory and possibly places the address in a general purpose register.

Syntax

stfdu *FRS,D(RA)*



Description

The **stfdu** instruction stores the contents of Floating Point Register *FRS* into the doubleword storage addressed by the effective address (EA).

The effective address (EA) is the sum of the contents of General Purpose Register *RA* and *D*, a 16-bit signed two's complement integer sign extended to 32 bits, if *RA* is not 0. If *RA* is 0, then the effective address (EA) is *D*.

- If *RA* does not equal 0 and the storage access does not cause Alignment Interrupt or a Data Storage Interrupt, then the effective address is stored in General Purpose Register RA.
- If alignment checking is disabled, i.e., the alignment bit (AL) in the Machine Status Register (MSR) is 0, then the three low order bits of the effective address are ignored.
- If alignment checking is enabled, and the three low order bits are not b'000', then the hardware attempts to perform the unaligned storage access. If the hardware cannot perform the unaligned storage access, an Alignment Interrupt is generated.

The **stfdu** instruction has one syntax form and does not affect the Fixed Point Exception Register or Condition Register Field 0.

Parameters

FRS	Specifies source floating point register of stored data.
D	16-bit signed two's complement integer sign extended to 32 bits for EA calculation.
RA	Specifies source general purpose register for EA calculation and possible address update.

Examples

1. To store the double word contents of FPR 6 into a location in memory and store the address in GPR 4:

```
.csect data[rw]
buffer: .long 0,0
# Assume FPR 6 contains 0x4865 6C6C 6F20 776F.
# GPR 4 contains the address of csect data[rw].
.csect text[pr]
stdfu 6,buffer(4)
# buffer now contains 0x4865 6C6C 6F20 776F.
# GPR 4 now contains the address of buffer.
```

Implementation Specifics

This instruction is part of Application Development Toolkit in AIX Base.

Related Information

Understanding Floating Point Store Instructions on page 1–13.

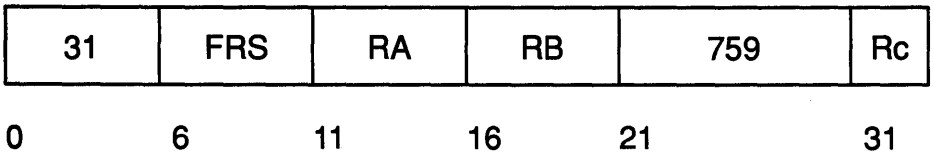
stfdux (Store Floating Point Double With Update Indexed) Instruction

Purpose

Stores a double word of data in a specified location in memory and possibly places the address in a general purpose register.

Syntax

stfdux *FRS,RA,RB*



Description

The **stfdux** instruction stores the contents of Floating Point Register *FRS* into the doubleword storage addressed by the effective address (EA).

The effective address (EA) is the sum of the contents of General Purpose Register *RA* and General Purpose Register *RB*, if *RA* is not 0. If *RA* is 0, then the effective address (EA) is the contents of *RB*.

- If *RA* does not equal 0 and the storage access does not cause Alignment Interrupt or a Data Storage Interrupt, then the effective address is stored in General Purpose Register *RA*.
- If alignment checking is disabled, i.e., the alignment bit (AL) in the Machine Status Register (MSR) is 0, then the three low order bits of the effective address are ignored.
- If alignment checking is enabled, and the three low order bits are not b'000', then the hardware attempts to perform the unaligned storage access. If the hardware cannot perform the unaligned storage access, an Alignment Interrupt is generated.

The **stfdux** instruction has one syntax form and does not affect the Fixed Point Exception Register or Condition Register Field 0.

Parameters

FRS	Specifies source floating point register of stored data.
RA	Specifies source general purpose register for EA calculation and possible address update.
RB	Specifies source general purpose register for EA calculation.

Examples

1. To store the contents of FPR 6 into a location in memory and store the address in GPR 4:

```
.csect data[rw]
buffer: .long 0,0,0,0
# Assume FPR 6 contains 0x9000 3000 9000 3000.
# Assume GPR 4 contains 0x0000 0008.
# Assume GPR 5 contains the address of buffer.
.csect text[pr]
stfdux 6,4,5
# buffer+8 now contains 0x9000 3000 9000 3000.
# GPR 4 now contains the address of buffer+8.
```

Implementation Specifics

This instruction is part of Application Development Toolkit in AIX Base.

Related Information

Understanding Floating Point Store Instructions on page 1–13.

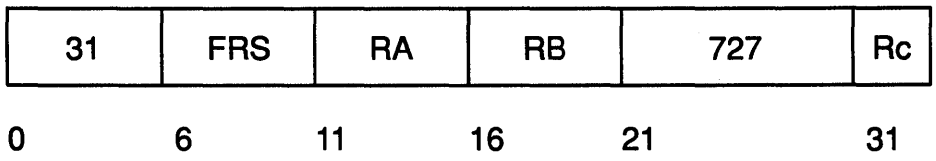
stfdx (Store Floating Point Double Indexed) Instruction

Purpose

Stores a double word of data in a specified location in memory.

Syntax

stfdx *FRS,RA,RB*



Description

The **stfdx** instruction stores the contents of Floating Point Register *FRS* into the doubleword storage addressed by the effective address (EA).

The effective address (EA) is the sum of the contents of General Purpose Register *RA* and General Purpose Register *RB* if *RA* is not 0. If *RA* is 0, then the effective address (EA) is the contents of General Purpose Register *RB*.

- If alignment checking is disabled, i.e., the alignment bit (AL) in the Machine Status Register (MSR) is 0, then the three low order bits of the effective address are ignored.
- If alignment checking is enabled, and the three low order bits are not b'000', then the hardware attempts to perform the unaligned storage access. If the hardware cannot perform the unaligned storage access, an Alignment Interrupt is generated.

The **stfdx** instruction has one syntax form and does not affect the Fixed Point Exception Register or Condition Register Field 0.

Parameters

FRS	Specifies source floating point register of stored data.
RA	Specifies source general purpose register for EA calculation.
RB	Specifies source general purpose register for EA calculation.

Examples

1. To store the contents of FPR 6 into a location in memory addressed by GPR 5 and GPR 4:

```
.csect data[rw]
buffer: .long 0,0,0,0
# Assume FPR 6 contains 0x4865 6C6C 6F20 776F.
# Assume GPR 4 contains 0x0000 0008.
# Assume GPR 5 contains the address of buffer.
.csect text[pr]
stfdx 6,4,5
# 0x4865 6C6C 6F20 776F is now stored at the
# address buffer+8.
```

Implementation Specifics

This instruction is part of Application Development Toolkit in AIX Base.

Related Information

Understanding Floating Point Store Instructions on page 1–13.

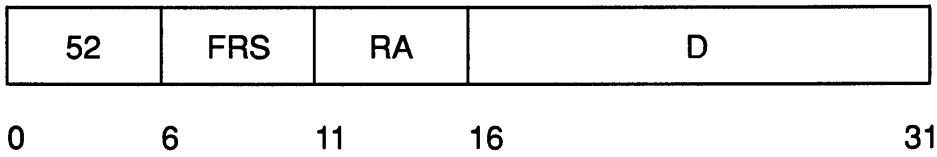
stfs (Store Floating Point Single) Instruction

Purpose

Stores a word of data from a floating point register into a specified location in memory.

Syntax

stfs *FRS,D(RA)*



Description

The **stfs** instruction converts the contents of Floating Point Register *FRS* to single precision and stores the result into the word of storage addressed by the effective address (EA).

The effective address (EA) is the sum of the contents of General Purpose Register *RA* and *D*, a 16-bit signed two's complement integer sign extended to 32 bits, if *RA* is not 0. If *RA* is 0, then the effective address (EA) is *D*.

- If alignment checking is disabled, i.e., the alignment bit (AL) in the Machine Status Register (MSR) is 0, then the two low order bits of the effective address are ignored.
- If alignment checking is enabled, and the two low order bits are not b'00', then the hardware attempts to perform the unaligned storage access. If the hardware cannot perform the unaligned storage access, an Alignment Interrupt is generated.

The **stfs** instruction has one syntax form and does not affect the Fixed Point Exception Register or Condition Register Field 0.

Parameters

FRS	Specifies floating point register of stored data.
D	16-bit signed two's complement integer sign extended to 32 bits for EA calculation.
RA	Specifies source general purpose register for EA calculation.

Examples

1. To store the single–precision contents of FPR 6 into a location in memory:

```
.csect data[rw]
buffer: .long 0
# Assume FPR 6 contains 0x4865 6C6C 6F20 776F.
# Assume GPR 4 contains the address of csect data[rw].
.csect text[pr]
stfs 6,buffer(4)
# buffer now contains 0x432B 6363.
```

Implementation Specifics

This instruction is part of Application Development Toolkit in AIX Base.

Related Information

Understanding Floating Point Store Instructions on page 1–13.

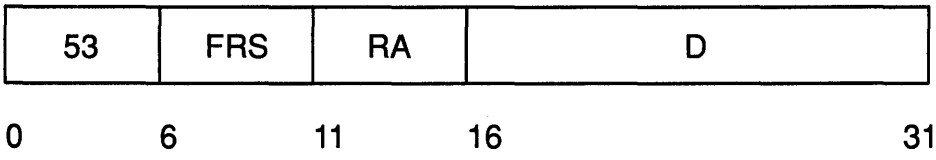
stfsu (Store Floating Point Single With Update) Instruction

Purpose

Stores a word of data from a floating point register into a specified location in memory and possibly places the address in a general purpose register.

Syntax

stfsu *FRS,D(RA)*



Description

The **stfsu** instruction converts the contents of Floating Point Register *FRS* to single precision and stores the result into the word of storage addressed by the effective address (EA).

The effective address (EA) is the sum of the contents of General Purpose Register *RA* and *D*, a 16-bit signed two's complement integer sign extended to 32 bits, if *RA* is not 0. If *RA* is 0, then the effective address (EA) is *D*.

- If *RA* does not equal 0 and the storage access does not cause Alignment Interrupt or Data Storage Interrupt, then the effective address is stored in General Purpose Register RA.
- If alignment checking is disabled, i.e., the alignment bit (AL) in the Machine Status Register (MSR) is 0, then the two low order bits of the effective address are ignored.
- If alignment checking is enabled, and the two low order bits are not b'00', then the hardware attempts to perform the unaligned storage access. If the hardware cannot perform the unaligned storage access, an Alignment Interrupt is generated.

The **stfsu** instruction has one syntax form and does not affect the Fixed Point Exception Register or Condition Register Field 0.

Parameters

FRS	Specifies floating point register of stored data.
D	16-bit signed two's complement integer sign extended to 32 bits for EA calculation.
RA	Specifies source general purpose register for EA calculation and possible address update.

Examples

1. To store the single–precision contents of FPR 6 into a location in memory and store the address in GPR 4:

```
.csect data[rw]
buffer: .long 0
```

```
# Assume FPR 6 contains 0x4865 6C6C 6F20 776F.
# Assume GPR 4 contains the address of csect data[rw].
.csect text[pr]
stfsu 6,buffer(4)
# GPR 4 now contains the address of buffer.
# buffer now contains 0x432B 6363.
```

Implementation Specifics

This instruction is part of Application Development Toolkit in AIX Base.

Related Information

Understanding Floating Point Store Instructions on page 1–13.

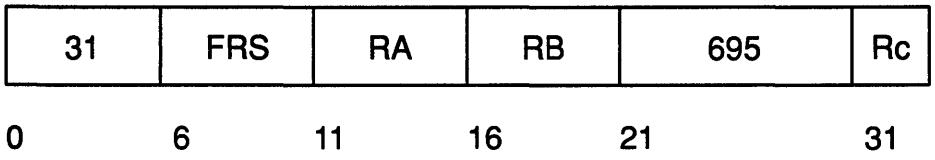
stfsux (Store Floating Point Single With Update Indexed) Instruction

Purpose

Stores a word of data from a floating point register into a specified location in memory and possibly places the address in a general purpose register.

Syntax

stfsux *FRS,RA,RB*



Description

The **stfsux** instruction converts the contents of Floating Point Register *FRS* to single precision and stores the result into the word of storage addressed by the effective address (EA).

The effective address (EA) is the sum of the contents of General Purpose Register *RA* and General Purpose Register *RB*, if *RA* is not 0. If *RA* is 0, then the effective address (EA) is the contents of *RB*.

- If *RA* does not equal 0 and the storage access does not cause Alignment Interrupt or Data Storage Interrupt, then the effective address is stored in General Purpose Register RA.
- If alignment checking is disabled, i.e., the alignment bit (AL) in the Machine Status Register (MSR) is 0, then the two low order bits of the effective address are ignored.
- If alignment checking is enabled, and the two low order bits are not b'00', then the hardware attempts to perform the unaligned storage access. If the hardware cannot perform the unaligned storage access, an Alignment Interrupt is generated.

The **stfsux** instruction has one syntax form and does not affect the Fixed Point Exception Register or Condition Register Field 0.

Parameters

FRS	Specifies floating point register of stored data.
RA	Specifies source general purpose register for EA calculation and possible address update.
RB	Specifies source general purpose register for EA calculation.

Examples

1. To store the single–precision contents of FPR 6 into a location in memory and store the address in GPR 5:

```
.csect data[rw]
buffer: .long 0,0,0,0
# Assume GPR 4 contains 0x0000 0008.
# Assume GPR 5 contains the address of buffer.
# Assume FPR 6 contains 0x4865 6C6C 6F20 776F.
.csect text[pr]
stfsux 6,5,4
# GPR 5 now contains the address of buffer+8.
# buffer+8 contains 0x432B 6363.
```

Implementation Specifics

This instruction is part of Application Development Toolkit in AIX Base.

Related Information

Understanding Floating Point Store Instructions on page 1–13.

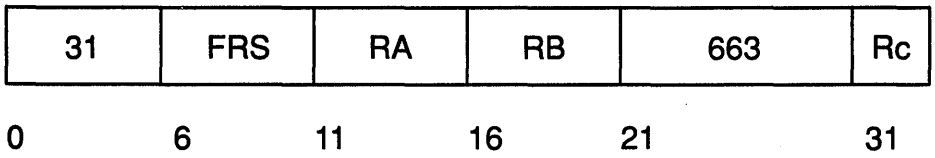
stfsx (Store Floating Point Single Indexed) Instruction

Purpose

Stores a word of data from a floating point register into a specified location in memory.

Syntax

stfsx *FRS,RA,RB*



Description

The **stfsx** instruction converts the contents of Floating Point Register *FRS* to single precision and stores the result into the word of storage addressed by the effective address (EA).

The effective address (EA) is the sum of the contents of General Purpose Register *RA* and General Purpose Register *RB*, if *RA* is not 0. If *RA* is 0, then the effective address (EA) is the contents of *RB*.

- If alignment checking is disabled, i.e., the alignment bit (AL) in the Machine Status Register (MSR) is 0, then the two low order bits of the effective address are ignored.
- If alignment checking is enabled, and the two low order bits are not b'00', then the hardware attempts to perform the unaligned storage access. If the hardware cannot perform the unaligned storage access, an Alignment Interrupt is generated.

The **stfsx** instruction has one syntax form and does not affect the Fixed Point Exception Register or Condition Register Field 0.

Parameters

FRS	Specifies source floating point register of stored data.
RA	Specifies source general purpose register for EA calculation.
RB	Specifies source general purpose register for EA calculation.

Examples

1. To store the single–precision contents of FPR 6 into a location in memory:

```
.csect data[rw]
buffer: .long 0
# Assume FPR 6 contains 0x4865 6C6C 6F20 776F.
# Assume GPR 4 contains the address of buffer.
.csect text[pr]
stfsx 6,0,4
# buffer now contains 0x432B 6363.
```

Implementation Specifics

This instruction is part of Application Development Toolkit in AIX Base.

Related Information

Understanding Floating Point Store Instructions on page 1–13.

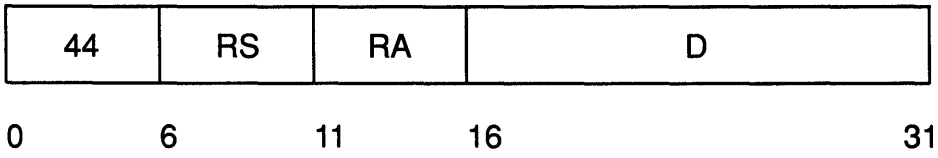
sth (Store Half) Instruction

Purpose

Stores a halfword of data from a general purpose register into a specified location in memory.

Syntax

sth *RS,D(RA)*



Description

The **sth** instruction stores bits 16–31 of General Purpose Register *RS* into the halfword of storage addressed by the effective address (EA).

The effective address (EA) is the sum of the contents of General Purpose Register *RA* and *D*, a 16-bit signed two's complement integer sign extended to 32 bits, if *RA* is not 0. If *RA* is 0, then the effective address (EA) is *D*.

- If alignment checking is disabled, i.e., the alignment bit (AL) in the Machine Status Register (MSR) is 0, then the low order bit of the effective address is ignored.
- If alignment checking is enabled, and the low order bit is not b'0', then the hardware attempts to perform the unaligned storage access. If the hardware cannot perform the unaligned storage access, an Alignment Interrupt is generated.

The **sth** instruction has one syntax form and does not affect the Fixed Point Exception Register or Condition Register Field 0.

Parameters

RS	Specifies source general purpose register of stored data.
D	16-bit signed two's complement integer sign extended to 32 bits for EA calculation.
RA	Specifies source general purpose register for EA calculation.

Examples

1. To store bits 16–31 of GPR 6 into a location in memory:

```
.csect data[rw]
buffer: .long 0
# Assume GPR 4 contains the address of csect data[rw].
# Assume GPR 6 contains 0x9000 3000.
.csect text[pr]
sth 6,buffer(4)
# buffer now contains 0x3000.
```

Implementation Specifics

This instruction is part of Application Development Toolkit in AIX Base.

Related Information

Understanding Fixed Point Store Instructions on page 1–7.

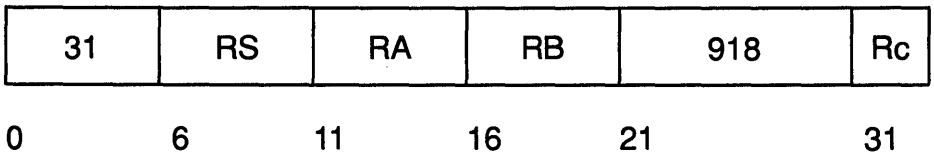
sthbrx (Store Half Byte Reverse Indexed) Instruction

Purpose

Stores a halfword of data from a general purpose register into a specified location in memory with the two bytes reversed.

Syntax

sthbrx *RS,RA,RB*



Description

The **sthbrx** instruction stores bits 16–31 of General Purpose Register *RS* into the halfword of storage addressed by the effective address (EA).

- Bits 24–31 of register *RS* are stored into bits 00–07 of the halfword in storage addressed by EA.
- Bits 16–23 of register *RS* are stored into bits 08–15 of the word in storage addressed by EA.

The effective address (EA) is the sum of the contents of General Purpose Register *RA* and General Purpose Register *RB* if *RA* is not 0. If *RA* is 0, then the effective address (EA) is the contents of *RB*.

- If alignment checking is disabled, i.e., the alignment bit (AL) in the Machine Status Register (MSR) is 0, then the low order bit of the effective address is ignored.
- If alignment checking is enabled, and the low order bit is not b'0', then the hardware attempts to perform the unaligned storage access. If the hardware cannot perform the unaligned storage access, an Alignment Interrupt is generated.

The **sthbrx** instruction has one syntax form and does not affect the Fixed Point Exception Register or Condition Register Field 0.

Parameters

RS	Specifies source general purpose register of stored data.
RA	Specifies source general purpose register for EA calculation.
RB	Specifies source general purpose register for EA calculation.

Examples

1. To store the halfword contents of GPR 6 with the bytes reversed into a location in memory:

```
.csect data[rw]
buffer: .long 0
# Assume GPR 6 contains 0x9000 3456.
# Assume GPR 4 contains the address of buffer.
.csect text[pr]
sthbrx 6,0,4
# buffer now contains 0x5634.
```

Implementation Specifics

This instruction is part of Application Development Toolkit in AIX Base.

Related Information

Understanding Fixed Point Store Instructions on page 1–7.

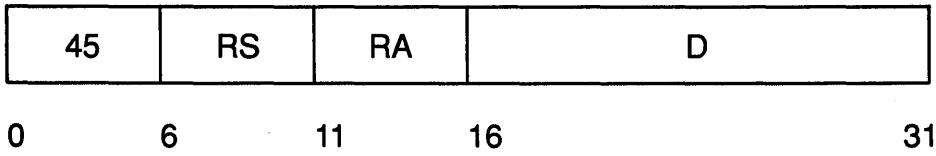
sthu (Store Half With Update) Instruction

Purpose

Stores a halfword of data from a general purpose register into a specified location in memory and possibly places the address in a general purpose register.

Syntax

sthu *RS,D(RA)*



Description

The **sthu** instruction stores bits 16–31 of General Purpose Register *RS* into the halfword of storage addressed by the effective address (EA).

The effective address (EA) is the sum of the contents of General Purpose Register *RA* and *D*, a 16-bit signed two's complement integer sign extended to 32 bits, if *RA* is not 0. If *RA* is 0, then the effective address (EA) is *D*.

- If *RA* does not equal 0 and the storage access does not cause an Alignment Interrupt or a Data Storage Interrupt, then the effective address is placed into register *RA*.
- If alignment checking is disabled, i.e., the alignment bit (AL) in the Machine Status Register (MSR) is 0, then the low order bit of the effective address is ignored.
- If alignment checking is enabled, and the low order bit is not b'0', then the hardware attempts to perform the unaligned storage access. If the hardware cannot perform the unaligned storage access, an Alignment Interrupt is generated.

The **sthu** instruction has one syntax form and does not affect the Fixed Point Exception Register or Condition Register Field 0.

Parameters

RS	Specifies source general purpose register of stored data.
D	16-bit signed two's complement integer sign extended to 32 bits for EA calculation.
RA	Specifies source general purpose register for EA calculation and possible address update.

Examples

1. To store the halfword contents of GPR 6 into a memory location and store the address in GPR 4:

```
.csect data[rw]
buffer: .long 0
# Assume GPR 6 contains 0x9000 3456.
# Assume GPR 4 contains the address of csect data[rw].
.csect text[pr]
sthu 6,buffer(4)
# buffer now contains 0x3456
# GPR 4 contains the address of buffer.
```

Implementation Specifics

This instruction is part of Application Development Toolkit in AIX Base.

Related Information

Understanding Fixed Point Store with Update Instructions on page 1–7.

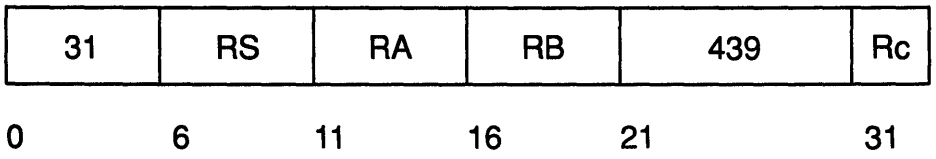
sthux (Store Half With Update Indexed) Instruction

Purpose

Stores a halfword of data from a general purpose register into a specified location in memory and possibly places the address in a general purpose register.

Syntax

sthux *RS,RA,RB*



Description

The **sthux** instruction stores bits 16–31 of General Purpose Register *RS* into the halfword of storage addressed by the effective address (EA).

The effective address (EA) is the sum of the contents of General Purpose Register *RA* and General Purpose Register *RB* if *RA* is not 0. If *RA* is 0, then the effective address (EA) is the contents of *RB*.

- If *RA* does not equal 0 and the storage access does not cause an Alignment Interrupt or a Data Storage Interrupt, then the effective address is placed into register *RA*.
- If alignment checking is disabled, i.e., the alignment bit (AL) in the Machine Status Register (MSR) is 0, then the low order bit of the effective address is ignored.
- If alignment checking is enabled, and the low order bit is not b'0', then the hardware attempts to perform the unaligned storage access. If the hardware cannot perform the unaligned storage access, an Alignment Interrupt is generated.

The **sthux** instruction has one syntax form and does not affect the Fixed Point Exception Register or Condition Register Field 0.

Parameters

RS	Specifies source general purpose register of stored data.
RA	Specifies source general purpose register for EA calculation and possible address update.
RB	Specifies source general purpose register for EA calculation.

Examples

1. To store the halfword contents of GPR 6 into a memory location and store the address in GPR 4:

```
.csect data[rw]
buffer: .long 0,0,0,0
# Assume GPR 6 contains 0x9000 3456.
# Assume GPR 4 contains 0x0000 0007.
# Assume GPR 5 contains the address of buffer.
.csect text[pr]
sthux 6,4,5
# buffer+0x07 contains 0x3456.
# GPR 4 contains the address of buffer+0x07.
```

Implementation Specifics

This instruction is part of Application Development Toolkit in AIX Base.

Related Information

Understanding Fixed Point Store with Update Instructions on page 1–7.

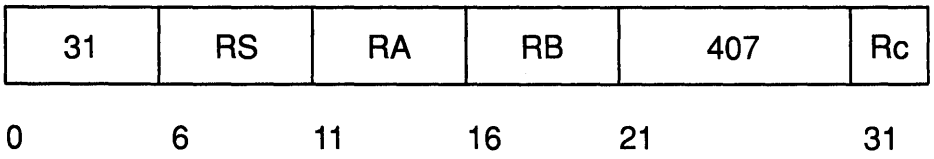
sthx (Store Half Indexed) Instruction

Purpose

Stores a halfword of data from a general purpose register into a specified location in memory.

Syntax

sthx *RS,RA,RB*



Description

The **sthx** instruction stores bits 16–31 of General Purpose Register *RS* into the halfword of storage addressed by the effective address (EA).

The effective address (EA) is the sum of the contents of General Purpose Register *RA* and General Purpose Register *RB* if *RA* is not 0. If *RA* is 0, then the effective address (EA) is the contents of *RB*.

- If alignment checking is disabled, i.e., the alignment bit (AL) in the Machine Status Register (MSR) is 0, then the low order bit of the effective address is ignored.
- If alignment checking is enabled, and the low order bit is not b'0', then the hardware attempts to perform the unaligned storage access. If the hardware cannot perform the unaligned storage access, an Alignment Interrupt is generated.

The **sthx** instruction has one syntax form and does not affect the Fixed Point Exception Register or Condition Register Field 0.

Parameters

RS	Specifies source general purpose register of stored data.
RA	Specifies source general purpose register for EA calculation.
RB	Specifies source general purpose register for EA calculation.

Examples

1. To store halfword contents of GPR 6 into a location in memory:

```
.csect data[rw]
buffer: .long 0
# Assume GPR 6 contains 0x9000 3456.
# Assume GPR 5 contains the address of buffer.
.csect text[pr]
sthx 6,0,5
# buffer now contains 0x3456.
```

Implementation Specifics

This instruction is part of Application Development Toolkit in AIX Base.

Related Information

Understanding Fixed Point Store Instructions on page 1–7.

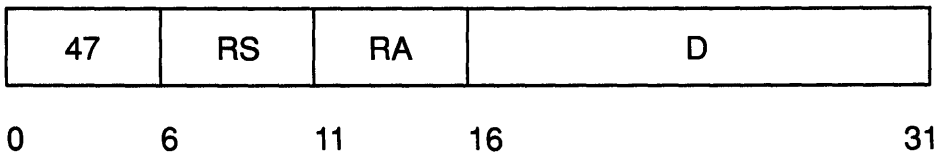
stm (Store Multiple) Instruction

Purpose

Stores the contents of consecutive registers into a specified memory location.

Syntax

stm *RS,D(RA)*



Description

The **stm** instruction stores *N* consecutive words from General Purpose Register *RS* through General Purpose Register 31. Storage starts at the effective address (EA). *N* is a register number equal to 32 minus *RS*.

The effective address (EA) is the sum of the contents of General Purpose Register *RA* and *D*, a 16-bit signed two's complement integer sign extended to 32 bits, if *RA* is not 0. If *RA* is 0, then the effective address (EA) is *D*.

- If alignment checking is disabled, i.e., the alignment bit (AL) in the Machine Status Register (MSR) is 0, then the two low order bits of the effective address are ignored.
- If alignment checking is enabled, and the two low order bits are not b'00', then the hardware attempts to perform the unaligned storage access. If the hardware cannot perform the unaligned storage access, an Alignment Interrupt is generated.

The **stm** instruction has one syntax form and does not affect the Fixed Point Exception Register or Condition Register Field 0.

Parameters

RS	Specifies source general purpose register of stored data.
D	16-bit signed two's complement integer sign extended to 32 bits for EA calculation.
RA	Specifies source general purpose register for EA calculation.

Examples

1. To store the contents of GPR 29 through GPR 31 into a location in memory:

```
.csect data[rw]
buffer: .long 0,0,0
# Assume GPR 29 contains 0x1000 2200.
# Assume GPR 30 contains 0x1000 3300.
# Assume GPR 31 contains 0x1000 4400.
.csect text[pr]
stm 29,buffer(4)
# Three consecutive words in storage beginning at the address
# of buffer are now 0x1000 2200 1000 3300 1000 4400.
```

Implementation Specifics

This instruction is part of Application Development Toolkit in AIX Base.

Related Information

Understanding Fixed Point Store Instructions on page 1–7.

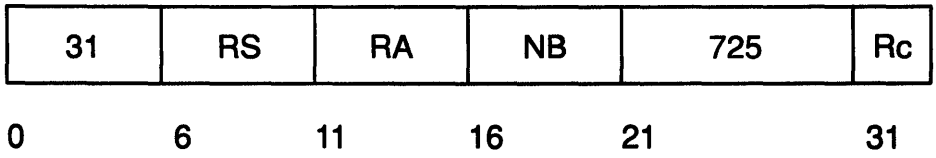
stsi (Store String Immediate) Instruction

Purpose

Stores consecutive bytes from consecutive registers into a specified location in memory.

Syntax

stsi *RS,RA,NB*



Description

The **stsi** instruction stores *N* consecutive bytes starting with the leftmost byte in register *RS* at the effective address (EA) from General Purpose Register *RS* through register *RS* + *NR* – 1.

The effective address (EA) is the contents of General Purpose Register *RA* if *RA* is not 0. If *RA* is 0, then the effective address (EA) is 0.

- *NB* is the byte count.
- *RS* is the starting register.
- *N* is *NB*, which is the number of bytes to store. If *NB* is 0, then *N* is 32.
- *NR* is ceiling(N/4), which is the number of registers to store data from.
- The contents of the MQ Register are undefined.

The **stsi** instruction has one syntax form and does not affect the Fixed Point Exception Register or Condition Register Field 0.

Parameters

RS	Specifies source general purpose register of stored data.
RA	Specifies source general purpose register for EA calculation.
NB	Specifies byte count for EA calculation.

Examples

1. To store the bytes contained in GPR 6 to GPR 8 into a location in memory:

```
.csect data[rw]
buffer: .long 0,0,0
# Assume GPR 4 contains the address of buffer.
# Assume GPR 6 contains 0x4865 6C6C.
# Assume GPR 7 contains 0x6F20 776F.
# Assume GPR 8 contains 0x726C 6421.
.csect text[pr]
stsi 6,4,12
# buffer now contains 0x4865 6C6C 6F20 776F 726C 6421.
```

Implementation Specifics

This instruction is part of Application Development Toolkit in AIX Base.

Related Information

Understanding Fixed Point String Instructions on page 1–8.

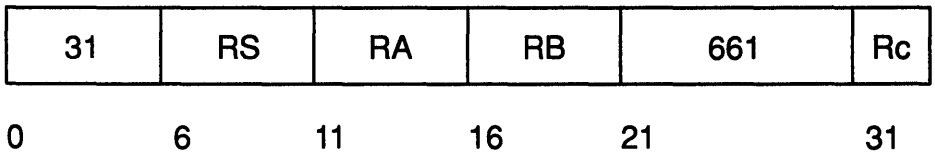
stsx (Store String Indexed) Instruction

Purpose

Stores consecutive bytes from consecutive registers into a specified location in memory.

Syntax

stsx *RS,RA,RB*



Description

The **stsx** instruction stores *N* consecutive bytes starting with the leftmost byte in register *RS* at the effective address (EA) from General Purpose Register *RS* through register *RS* + *NR* – 1.

The effective address (EA) is the sum of the contents of General Purpose Register *RA* and the contents of General Purpose Register *RB* if *RA* is not 0. If *RA* is 0, then effective address (EA) is the contents of *RB*.

- *XER25–31* contain the byte count.
- *RS* is the starting register.
- *N* is *XER25–31*, which is the number of bytes to store.
- *NR* is ceiling(N/4), which is the number of registers to store data from.
- The contents of the MQ Register are undefined.

The **stsx** instruction has one syntax form and does not affect the Fixed Point Exception Register or Condition Register Field 0.

Parameters

RS	Specifies source general purpose register of stored data.
RA	Specifies source general purpose register for EA calculation.
RB	Specifies source general purpose register for EA calculation.

Examples

1. To store the bytes contained in GPR 6 to GPR 7 into the specified bytes of a location in memory:

```
.csect data[rw]
buffer: .long 0,0,0
# Assume GPR 5 contains 0x0000 0007.
# Assume GPR 4 contains the address of buffer.
# Assume GPR 6 contains 0x4865 6C6C.
# Assume GPR 7 contains 0x6F20 776F.
# The Fixed Point Exception Register bits 25–31 contain 6.
.csect text[pr]
stsx 6,4,5
# buffer+0x7 now contains 0x4865 6C6C 6F20.
```

Implementation Specifics

This instruction is part of Application Development Toolkit in AIX Base.

Related Information

Understanding Fixed Point String Instructions on page 1–8.

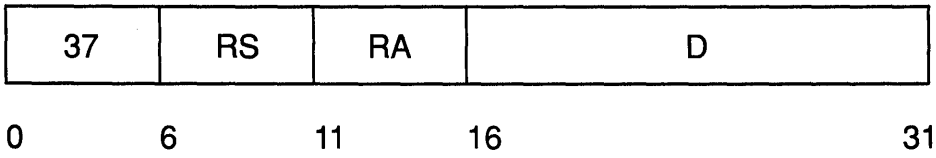
stu (Store With Update) Instruction

Purpose

Stores a word of data from a general purpose register into a specified location in memory and possibly places the address in a general purpose register.

Syntax

stu *RS,D(RA)*



Description

The **stu** instruction stores the contents of General Purpose Register *RS* into the word of storage addressed by the effective address (EA).

The effective address (EA) is the sum of the contents of General Purpose Register *RA* and *D*, a 16-bit signed two's complement integer sign extended to 32 bits, if *RA* is not 0. If *RA* is 0, then the effective address (EA) is *D*.

- If *RA* is not 0 and the storage access does not cause an Alignment Interrupt or a Data Storage Interrupt, then EA is placed into register RA.
- If alignment checking is disabled, i.e., the alignment bit (AL) in the Machine Status Register (MSR) is 0, then the two low order bits of the effective address are ignored.
- If alignment checking is enabled, and the two low order bits are not b'00', then the hardware attempts to perform the unaligned storage access. If the hardware cannot perform the unaligned storage access, an Alignment Interrupt is generated.

The **stu** instruction has one syntax form and does not affect the Fixed Point Exception Register or Condition Register Field 0.

Parameters

RS	Specifies general purpose register of stored data.
D	16-bit signed two's complement integer sign extended to 32 bits for EA calculation.
RA	Specifies source general purpose register for EA calculation and possible address update.

Examples

1. To store the contents of GPR 6 into a location in memory:

```
.csect data[rw]
buffer: .long 0
# Assume GPR 4 contains the address of csect data[rw].
# Assume GPR 6 contains 0x9000 3000.
.csect text[pr]
stu 6,buffer(4)
# buffer now contains 0x9000 3000.
# GPR 4 contains the address of buffer.
```

Implementation Specifics

This instruction is part of Application Development Toolkit in AIX Base.

Related Information

Understanding Fixed Point Store with Update Instructions on page 1–7.

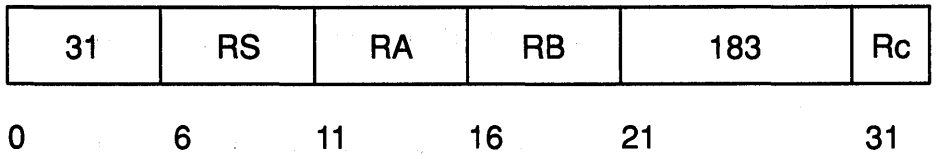
stux (Store with Update Indexed) Instruction

Purpose

Stores a word of data from a general purpose register into a specified location in memory and possibly places the address in a general purpose register.

Syntax

stux *RS,RA,RB*



Description

The **stux** instruction stores the contents of General Purpose Register *RS* into the word of storage addressed by the effective address (EA).

The effective address (EA) is the sum of the contents of General Purpose Register *RA* and General Purpose Register *RB* if *RA* is not 0. If *RA* is 0, then the effective address (EA) is the contents of *RB*.

- If *RA* is not 0 and the storage access does not cause an Alignment Interrupt or a Data Storage Interrupt, then EA is placed into register RA.
- If alignment checking is disabled, i.e., the alignment bit (AL) in the Machine Status Register (MSR) is 0, then the two low order bits of the effective address are ignored.
- If alignment checking is enabled, and the two low order bits are not b'00', then the hardware attempts to perform the unaligned storage access. If the hardware cannot perform the unaligned storage access, an Alignment Interrupt is generated.

The **stux** instruction has one syntax form and does not affect the Fixed Point Exception Register or Condition Register Field 0.

Parameters

RS	Specifies source general purpose register of stored data.
RA	Specifies source general purpose register for EA calculation and possible address update.
RB	Specifies source general purpose register for EA calculation.

Examples

1. To store the contents of GPR 6 into a location in memory:

```
.csect data[rw]
buffer: .long 0,0
# Assume GPR 4 contains 0x0000 0004.
# Assume GPR 23 contains the address of buffer.
# Assume GPR 6 contains 0x9000 3000.
.csect text[pr]
stux 6,4,23
# buffer+4 now contains 0x9000 3000.
# GPR 4 now contains the address of buffer+4.
```

Implementation Specifics

This instruction is part of Application Development Toolkit in AIX Base.

Related Information

Understanding Fixed Point Store with Update Instructions on page 1–7.

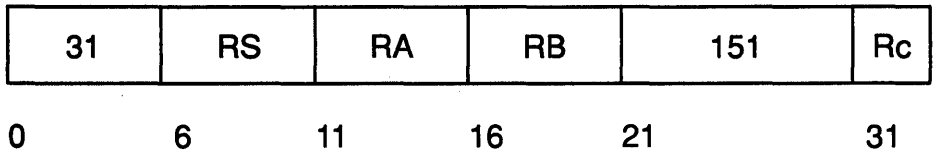
stx (Store Indexed) Instruction

Purpose

Stores a word of data from a general purpose register into a specified location in memory.

Syntax

stx *RS,RA,RB*



Description

The **stx** instruction stores the contents of General Purpose Register *RS* into the word of storage addressed by the effective address (EA).

The effective address (EA) is the sum of the contents of General Purpose Register *RA* and General Purpose Register *RB* if *RA* is not 0. If *RA* is 0, then the effective address (EA) is the contents of *RB*.

- If alignment checking is disabled, i.e., the alignment bit (AL) in the Machine Status Register (MSR) is 0, then the two low order bits of the effective address are ignored.
- If alignment checking is enabled, and the two low order bits are not b'00', then the hardware attempts to perform the unaligned storage access. If the hardware cannot perform the unaligned storage access, an Alignment Interrupt is generated.

The **stx** instruction has one syntax form and does not affect the Fixed Point Exception Register Condition Register Field 0.

Parameters

RS	Specifies source general purpose register of stored data.
RA	Specifies source general purpose register for EA calculation.
RB	Specifies source general purpose register for EA calculation.

Examples

1. To store the contents of GPR 6 into a location in memory:

```
.csect data[pr]
buffer: .long 0
# Assume GPR 4 contains the address of buffer.
# Assume GPR 6 contains 0x4865 6C6C.
.csect text[pr]
stx 6,0,4
# Buffer now contains 0x4865 6C6C.
```

Implementation Specifics

This instruction is part of Application Development Toolkit in AIX Base.

Related Information

Understanding Fixed Point Store Instructions on page 1–7.

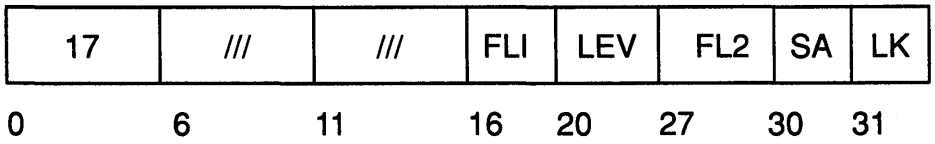
svc (Supervisor Call) Instruction

Purpose

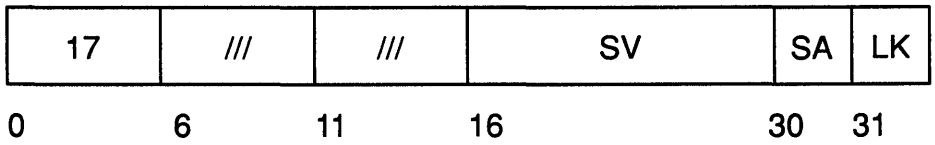
Generates a Supervisor Call interrupt.

Syntax

svc *LEV,FL1,FL2*
svcl *LEV,FL1,FL2*



svca *SV*
svcla *SV*



Description

The **svc** instruction generates a Supervisor Call interrupt and places bits 16–31 of the **svc** instruction into bits 0–15 of the Count Register and bits 16–31 of the Machine Status Register into bits 16–31 of the Count Register.

- If the SVC Absolute bit (SA) is set to zero, the instruction fetch and execution continues at one of the 128 offsets, b'1'|| LEV ||b'00000', to the base effective address indicated by the setting of the IP bit of the Machine State Register. *FL1* and *FL2* fields could be used for passing data to the SVC routine but are ignored by hardware.
- If the SVC Absolute bit (SA) is set to one, then instruction fetch and execution continues a the offset, x'1FE0', to the base effective address indicated by the setting of the IP bit of the Machine State Register.
- If the Link bit (LK) is set to one, the effective address of the instruction following the **svc** instruction is placed in the Link Register.

Note: To insure correct operation, an **svc** instruction must be preceded by an unconditional branch or a condition register instruction without an intervening conditional branch. If a useful instruction cannot be scheduled as specified, a no–op version of the **cror** instruction can be used.

cror *BT,BA,BB* No–op when *BT* = *BA* = *BB*

The **svc** instruction has four syntax forms. Each syntax form affects the Machine State Register.

Syntax form	Link bit (LK)	SVC Absolute bit (SA)	Machine State Register bits
svc	0	0	EE,PR,FE set to zero

svcl	1	0	EE,PR,FE set to zero
svca	0	1	EE,PR,FE set to zero
svcla	1	1	EE,PR,FE set to zero

The four syntax forms of the **svc** instruction never affect the FP, ME, AL, IP, IR, or DR bits of the Machine State Register. The EE, PR, and FE bits of the Machine State Register are always set to zero. The Fixed Point Exception Register and Condition Register Field 0 are unaffected by the **svc** instruction.

Parameters

LEV Specifies execution address.

FL1 Specifies field for optional data passing to SVC routine.

FL2 Specifies field for optional data passing to SVC routine.

SV Specifies field for optional data passing to SVC routine.

Implementation Specifics

This instruction is part of Application Development Toolkit in AIX Base.

Related Information

The **cror** (Condition Register OR) instruction.

Branch Processor Overview on page 1–3.

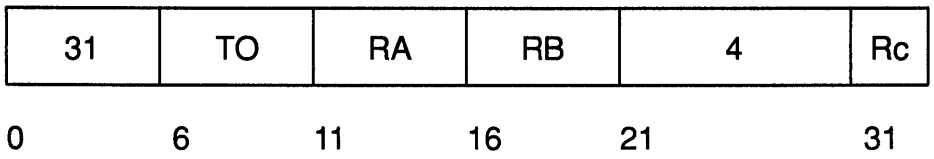
t (Trap) Instruction

Purpose

Generates a program interrupt when a specified condition is true.

Syntax

t *TO,RA,RB*



Description

The **t** instruction compares the contents of General Purpose Register *RA* with the contents of General Purpose Register *RB*, ANDs the compare results with *TO*, and generates a trap type Program Interrupt if the result is not 0.

The *TO* bit conditions are defined as follows.

TO bit	ANDed with Condition
6	Compares Less Than
7	Compares Greater Than
8	Compares Equal
9	Compares Logically Less Than
10	Compares Logically Greater Than

The **t** instruction has one syntax form and does not affect the Fixed Point Exception Register or Condition Register Field 0.

Parameters

TO	Specifies *TO* bits which are ANDed with compare results.
RA	Specifies source general purpose register for compare.
RB	Specifies source general purpose register for compare.

Extended Mnemonics

Eleven extended mnemonic trap instructions are based on the **t** (Trap) instruction. They are provided for commonly used traps.

Syntax	Parameters	Description
tlt	*RA,RB*	Trap if *RA* < *RB*
tgt	*RA,RB*	Trap if *RA* > *RB*
teq	*RA,RB*	Trap if *RA* = *RB*
tllt	*RA,RB*	Trap if *RA* logically < *RB*
tlgt	*RA,RB*	Trap if *RA* logically > *RB*

tle	*RA,RB*	Trap if *RA* < or = *RB*
tge	*RA,RB*	Trap if *RA* > or = *RB*
tne	*RA,RB*	Trap if *RA* not = *RB*
tlle	*RA,RB*	Trap if *RA* logically < or = *RB*
tlge	*RA,RB*	Trap if *RA* logically > or = *RB*
tlne	*RA,RB*	Trap if *RA* logically not = *RB*

Examples

1. To generate a trap type Program Interrupt:

```
# Assume GPR 4 contains 0x9000 3000.
# Assume GPR 7 contains 0x789A 789B.
t 0x10,4,7
# A trap type Program Interrupt occurs.
```

Implementation Specifics

This instruction is part of Application Development Toolkit in AIX Base.

Related Information

Understanding Trap Instructions on page 1–4.

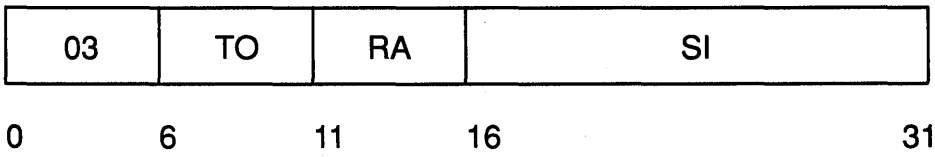
ti (Trap Immediate) Instruction

Purpose

Generates a program interrupt when a specified condition is true.

Syntax

ti *TO,RA,SI*



Description

The **ti** instruction compares the contents of General Purpose Register *RA* with the sign–extended *SI* field, ANDs the compare results with *TO*, and generates a trap type Program Interrupt if the result is not 0.

The *TO* bit conditions are defined as follows.

TO bit	ANDed with Condition
6	Compares Less Than
7	Compares Greater Than
8	Compares Equal
9	Compares Logically Less Than
10	Compares Logically Greater Than

The **ti** instruction has one syntax form and does not affect the Fixed Point Exception Register or Condition Register Field 0.

Parameters

TO	Specifies *TO* bits which are ANDed with compare results.
RA	Specifies source general purpose register for compare.
SI	Specifies sign extended value for compare.

Extended Mnemonics

Eleven extended mnemonic trap instructions are based on the **ti** (Trap Immediate) instruction. They are provided for commonly used traps.

Syntax	Parameters	Description
tlti	*RA,SI*	Trap if *RA* < *SI*
tgti	*RA,SI*	Trap if *RA* > *SI*
teqi	*RA,SI*	Trap if *RA* = *SI*
tllti	*RA,SI*	Trap if *RA* logically < *SI*
tlgti	*RA,SI*	Trap if *RA* logically > *SI*

tlei	*RA,SI*	Trap if *RA* < or = *SI*
tgei	*RA,SI*	Trap if *RA* > or = *SI*
tnei	*RA,SI*	Trap if *RA* not = *SI*
tllei	*RA,SI*	Trap if *RA* logically < or = *SI*
tlgei	*RA,SI*	Trap if *RA* logically > or = *SI*
tlnei	*RA,SI*	Trap if *RA* logically not = *SI*

Examples

1. To generate a Program Interrupt:

```
# Assume GPR 4 holds 0x0000 0010.
ti 0x4,4,0x10
# A trap type Program Interrupt occurs.
```

Implementation Specifics

This instruction is part of Application Development Toolkit in AIX Base.

Related Information

Understanding Trap Instructions on page 1–4.

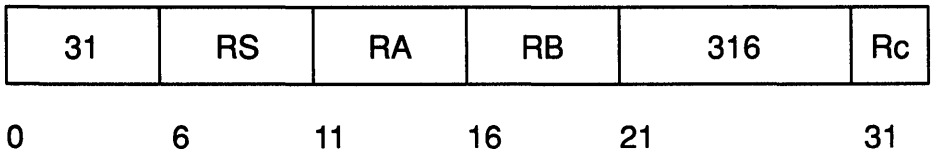
xor (XOR) Instruction

Purpose

XORs the contents of two general purpose registers and places the result in a general purpose register.

Syntax

xor *RA,RS,RB*
xor. *RA,RS,RB*



Description

The **xor** instruction XORs the contents of General Purpose Register *RS* with the contents of General Purpose Register *RB* and stores the result in General Purpose Register *RA*.

The **xor** instruction has two syntax forms. Each syntax form has a different effect on Condition Register Field 0.

Syntax form	Overflow Exception (OE)	Fixed Point Exception Register	Record bit (Rc)	Condition Register Field 0
xor	None	None	0	None
xor.	None	None	1	LT,GT,EQ,SO

The two syntax forms of the **xor** instruction never affect the Fixed Point Exception Register. If the syntax form sets the Record (Rc) bit to 1, the instruction affects the Less Than (LT) zero, Greater Than (GT) zero, Equal To (EQ) zero, and Summary Overflow (SO) bits in Condition Register Field 0.

Parameters

RA Specifies target general purpose register where result of operation is stored.

RS Specifies source general purpose register for operation.

RB Specifies source general purpose register for operation.

Examples

1. To XOR the contents of GPR 4 and GPR 7 and store the result in GPR 6:

```
# Assume GPR 4 contains 0x9000 3000.
# Assume GPR 7 contains 0x789A 789B.
xor 6,4,7
# GPR 6 now contains 0xE89A 489B.
```

2. To XOR the contents of GPR 4 and GPR 7,store the result in GPR 6, and set Condition Register Field 0 to reflect the result of the operation:

```
# Assume GPR 4 contains 0xB004 3000.
# Assume GPR 7 contains 0x789A 789B.
xor. 6,4,7
# GPR 6 now contains 0xC89E 489B.
```

Implementation Specifics

This instruction is part of Application Development Toolkit in AIX Base.

Related Information

Understanding Fixed Point Logical Instructions on page 1–9.

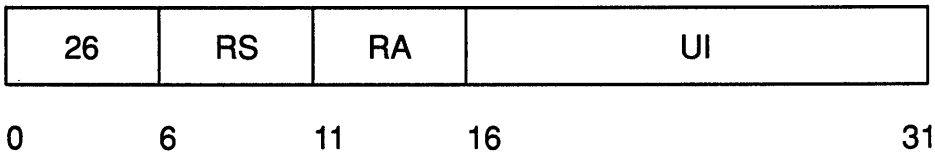
xoril (XOR Immediate Lower) Instruction

Purpose

XORs the lower 16 bits of a general purpose register with a 16-bit unsigned integer and places the result in a general purpose register.

Syntax

xoril *RA,RS,UI*



Description

The **xoril** instruction XORs the contents of General Purpose Register *RS* with the concatenation of x'0000' and a 16-bit unsigned integer *UI* and stores the result in General Purpose Register *RA*.

The **xoril** instruction has only one syntax form and does not affect the Fixed Point Exception Register or Condition Register Field 0.

Parameters

RA	Specifies target general purpose register where result of operation is stored.
RS	Specifies source general purpose register for operation.
UI	Specifies 16–bit unsigned integer for operation.

Examples

1. To XOR GPR 4 with 0x0000 5730, placing the result in GPR 6:

```
# Assume GPR 4 contains 0x7B41 92C0.
xoril 6,4,0x5730
# GPR 6 now contains 0x7B41 C5F0.
```

Implementation Specifics

This instruction is part of Application Development Toolkit in AIX Base.

Related Information

Understanding Fixed Point Logical Instructions on page 1–9.

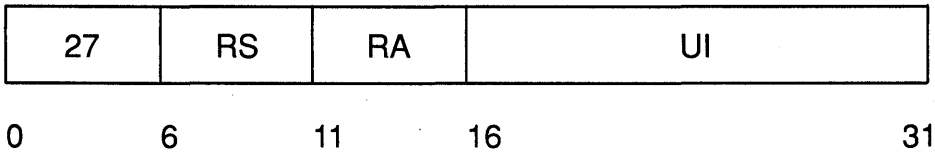
xoriu (XOR Immediate Upper) Instruction

Purpose

XORs the upper 16 bits of a general purpose register with a 16-bit unsigned integer and places the result in a general purpose register.

Syntax

xoriu *RA,RS,UI*



Description

The **xoriu** instruction XORs the contents of General Purpose Register *RS* with the concatenation of a 16-bit unsigned integer *UI* and 0x'0000' and stores the result in General Purpose Register *RA*.

The **xoriu** instruction has only one syntax form and does not affect the Fixed Point Exception Register or Condition Register Field 0.

Parameters

RA	Specifies target general purpose register where result of operation is stored.
RS	Specifies source general purpose register for operation.
UI	Specifies 16–bit unsigned integer for operation.

Examples

1. To XOR GPR 4 with 0x0079 0000 and store the result in GPR 6:

```
# Assume GPR 4 holds 0x9000 3000.
xoriu 6,4,0x0079
# GPR 6 now holds 0x9079 3000.
```

Implementation Specifics

This instruction is part of Application Development Toolkit in AIX Base.

Related Information

Understanding Fixed Point Logical Instructions on page 1–9.

Chapter 6. Pseudo–ops

Pseudo-ops Overview

A pseudo-operation, commonly called a pseudo-op, is an instruction to the assembler that does not generate any machine code. 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. The RISC System/6000 assembler also supports pseudo-ops that give the assembler information about floating point constants and symbolic debugger information (**dbx**).

While they do not generate machine code, the following pseudo-ops can change the contents of the assembler's location counter:

.align Pseudo–op

.byte Pseudo–op

.comm Pseudo–op

.csect Pseudo–op

.double Pseudo–op

.dsect Pseudo–op

.float Pseudo–op

.lcomm Pseudo–op

.long Pseudo–op

.org Pseudo–op

.short Pseudo–op

.space Pseudo–op

.string Pseudo–op

.vbyte Pseudo–op

Pseudo-ops can be related according to functionality into the following groups:

- Data alignment
- Data definition
- Storage definition
- Addressing within a source module (Base Registers)
- Direct addressing
- Assembler section definition
- External symbol definition
- Symbol table entries for debuggers

Data Alignment

The following pseudo-op is used in the data or text section of a program:

.align Pseudo–op

Data Definition

The following pseudo-ops are used for data definition:

.byte Pseudo–op

.double Pseudo–op

.float Pseudo–op

.long Pseudo–op

.short Pseudo–op

.string Pseudo–op

.vbyte Pseudo–op

In most instances these pseudo-ops create data areas to be used by a program.

```
        .csect data[rw]
greeting:       .long 'H,'O,'W,'D,'Y
        .
        .
        .csect text[pr]
                # Assume GPR 5 contains the address of
                # csect data[rw].
        lm 11, greeting(5)
```

Storage Definition

The following pseudo-ops define or map storage:

.dsect Pseudo–op

.space Pseudo–op

Addressing

The following pseudo–ops assign or dismiss a register as a base register:

.drop Pseudo–op

.using Pseudo–op

Assembler Section Definition

The following pseudo-ops define the sections of an assembly language program:

.comm Pseudo–op

.csect Pseudo–op

.lcomm Pseudo–op

.tc Pseudo–op

.toc Pseudo–op

External Symbol Definition

The following pseudo-ops define a variable as a global variable or an external variable (variables defined in external modules):

.extern Pseudo–op

.globl Pseudo–op

Support for Calling Conventions

The following pseudo–op defines a debug traceback tag for performing tracebacks when debugging programs:

.tbtag Pseudo–op

Symbol Table Entries for Debuggers

The following pseudo–ops provide additional information which is required by the symbolic debugger (**dbx**):

.bb Pseudo–op

.bc Pseudo–op

.bf Pseudo–op

.bi Pseudo–op

.bs Pseudo–op

.eb Pseudo–op

.ec Pseudo–op

.ef Pseudo–op

.ei Pseudo–op

.es Pseudo–op

.file Pseudo–op

.function Pseudo–op

.line Pseudo–op

.stabx Pseudo–op

.xline Pseudo–op

Miscellaneous

Other pseudo-ops set the value of the current location counter (**.org** Pseudo–op), give a value and type to a label (**.set** Pseudo–op) , create a synonym or alias for an illegal or undesirable name (**.rename** Pseudo–op), define a symbol as the TOC of another module (**.tocof** Pseudo–op), provide type checking information (**.hash** Pseudo–op), or provide file and line number information (**.xline** Pseudo–op).

Notational Conventions

White space is required unless otherwise specified. A space may optionally occur after a comma. White space may consist of one or more white spaces.

Some examples of pseudo-ops may not use labels. However, with the exception of **.csect**, 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:

Name	Any valid label.
Register	A General Purpose Register. *Register* is an expression that evaluates to an integer between 0 and 31 inclusive.
Number	An expression that evaluates to an integer.
Expression	Unless otherwise noted, *Expression* signifies a relocatable constant or absolute expression.
FloatingConstant	A floating point constant.
StringConstant	A string constant.
[ ]	Brackets enclose optional operands except in the **.csect** pseudo–op and **.tc** pseudo–op which require brackets in syntax.

.align Pseudo–op

Purpose

Advances the current location counter until a boundary specified by *Number* is reached.

Syntax

.align *Number*

Description

The **.align** pseudo-op is normally used in a control section (csect) which contains data.

If the *Number* parameter evaluates to 0, alignment occurs on a byte boundary. If the *Number* parameter evaluates to 1, alignment occurs on a halfword boundary. If the *Number* parameter evaluates to 2, alignment occurs on a word boundary. If the *Number* parameter evaluates to 3, alignment occurs on a doubleword boundary.

If the location counter is not aligned as specified by *Number*, the assembler advances the current location counter until the *Number* low–order bits are filled with the value 0 (zero).

If the **.align** pseudo-op is used within a **.csect** pseudo-op of type [PR] or [GL] which indicates a section containing instructions, alignment occurs by padding with **nop**, or no operation, instructions. In this instance, the no operation instruction is equivalent to a branch to the following instruction. If the align amount is less than a fullword, the padding consists of zeros.

Parameters

Number Specifies an expression that evaluates to the integer value of 0 (zero), 1, 2, or 3.

Example

```
                .csect  progdata[RW]
                .byte   1
                        # Location counter now at odd number
                .align  1

                        # Location counter is now at the next
                        # halfword boundary.
                .byte   3,4
                .

                .
                .
                .align  2       # Insure that the label cont
                                # and the .long pseudo–op are
aligned
                                # on a full word boundary.

        cont:   .long   5004381
```

Implementation Specifics

This pseudo–op is part of Application Development Toolkit in AIX Base.

Related Information

The **.byte** pseudo–op, **.comm** pseudo–op, **.csect** pseudo–op, **.double** pseudo–op, **.float** pseudo–op, **.long** pseudo–op, **.short** pseudo–op.

Pseudo–ops Overview on page 6–2 .

.bb Pseudo–op

Purpose

Identifies the beginning of an inner block and provides information specific to the beginning of an inner block.

Syntax

.bb *Number*

Description

The **.bb** pseudo–op provides symbol table information necessary for the use of the symbolic debugger and has no other effect on assembly.

This pseudo–op is customarily inserted by a compiler.

Parameters

Number Specifies the line number in the original source file on which the inner block begins.

Example

```
.bb     5
```

Implementation Specifics

This pseudo–op is part of Application Development Toolkit in AIX Base.

Related Information

The **.eb** pseudo–op.

Pseudo–ops Overview on page 6–2 .

.bc Pseudo–op

Purpose

Identifies the beginning of a common block and provides information specific to the beginning of a common block.

Syntax

.bc *StringConstant*

Description

The **.bc** pseudo–op provides symbol table information necessary for the use of the symbolic debugger and has no other effect on assembly.

This pseudo–op is customarily inserted by a compiler.

Parameters

StringConstant Represents the symbol name of the common block as defined in the original source file.

Example

```
.bc     "commonblock"
```

Implementation Specifics

This pseudo–op is part of Application Development Toolkit in AIX Base.

Related Information

The **.ec** pseudo–op.

Pseudo–ops Overview on page 6–2 .

.bf Pseudo–op

Purpose

Identifies the beginning of a function and provides information specific to the beginning of a function.

Syntax

.bf *Number*

Description

The **.bf** pseudo–op provides symbol table information necessary for the use of the symbolic debugger and has no other effect on assembly.

This pseudo–op is customarily inserted by a compiler.

Parameters

Number Represents the absolute line number in the original source file on which the function begins.

Example

```
.bf     5
```

Implementation Specifics

This pseudo–op is part of Application Development Toolkit in AIX Base.

Related Information

The **.ef** pseudo–op.

Pseudo–ops Overview on page 6–2 .

.bi Pseudo–op

Purpose

Identifies the beginning of an included file and provides information specific to the beginning of an included file.

Syntax

.bi *StringConstant*

Description

The **.bi** pseudo–op provides symbol table information necessary for the use of the symbolic debugger and has no other effect on assembly.

This pseudo–op is customarily inserted by a compiler.

Parameters

StringConstant Represents the name of the original source file.

Example

```
.bi     "file.s"
```

Implementation Specifics

This pseudo–op is part of Application Development Toolkit in AIX Base.

Related Information

The **.ei** Pseudo–op.

Pseudo–ops Overview on page 6–2 .

.bs Pseudo–op

Purpose

Identifies the beginning of a static block and provides information specific to the beginning of a static block.

Syntax

.bs *Name*

Description

The **.bs** pseudo–op provides symbol table information necessary for the use of the symbolic debugger and has no other effect on assembly.

This pseudo–op is customarily inserted by a compiler.

Parameters

Name Represents the symbol name of the static block as defined in the original source file.

Example

```
.lcomm cgdat, 0x2b4
.csect .text[PR]
.bs cgdat
        .stabx  "ONE:1=Ci2,0,4;",0x254,133,0
        .stabx  "TWO:S2=G5TWO1:3=Cc5,0,5;,0,40;;",0x258,133,8
.es
```

Implementation Specifics

This pseudo–op is part of Application Development Toolkit in AIX Base.

Related Information

The **.comm** pseudo–op, **.es** pseudo–op, **.lcomm** pseudo–op.

Pseudo–ops Overview on page 6–2 .

.byte Pseudo–op

Purpose

Assembles the values represented by *Expression* into consecutive bytes.

Syntax

.byte *Expression*[,*Expression*...]

Description

The **.byte** pseudo–op changes an *Expression* or a number of *Expressions* into consecutive bytes of data. ASCII character constants (for example, **'X**) and string constants (for example, "Hello, world") can also be assembled using **.byte**. Each letter will be assembled into consecutive bytes. However, an *Expression* cannot contain externally defined symbols, and if an *Expression* is longer than one byte, it will be truncated on the left.

Parameters

Expression Value which is assembled into consecutive bytes by instruction.

Example

```
        .set olddata,0xCC
        .csect data[rw]
mine:   .byte 0x3F,0x7+0xA,olddata,0xFF

# Load GPR 3 with the address of csect data[rw].
        .csect text[pr]
        l 3,mine(4)

# GPR 3 now holds 0x3F11 CCFF.
# Character constants can be represented in
# several ways:

        .csect data[rw]
        .byte "Hello, world"
        .byte 'H,'e,'l,'l,'o,',,' ,'w,'o,'r,'l,'d

# Both of the .byte statements will produce
# 0x4865 6C6C 6F2C 2077 6F72 6C64.
```

Implementation Specifics

This pseudo–op is part of Application Development Toolkit in AIX Base.

Related Information

The **.string** Pseudo–op, **.vbyte** Pseudo–op.

Pseudo–ops Overview on page 6–2 .

.comm Pseudo–op

Purpose

Defines an uninitialized block of storage called a common block which can be common to more than one module.

Syntax

.comm *Name,Expression*[,*Number*]

Description

The **.comm** pseudo–op initializes a block of data called *Name* when the size, which is defined in bytes in *Expression*, of the block is known but the block is not to be initialized locally. In other words, the block may not be initialized or may be initialized in another module.

Several modules can share the common block. If any of those modules have an external Control Section (csect) with the same name and a different storage class, then the common block is assumed to be initialized and becomes that other Control Section. Otherwise, the block is a Control Section of storage class **RW** (Read Write) and storage type **CM** (Common). At load time, the space for **CM** Control Sections is created in the .bss section at the end of the .data section.

If more than one uninitialized common block with the same name is found at bind time, space is reserved for the largest one.

A common block can be aligned by using *Number*, which is specified as the log base 2 of the alignment desired. For example, an alignment of 8 (or doubleword) would be 3 and an alignment of 2048 would be 11. This is similar to the argument for the **.align** pseudo-op.

Parameters

Name	Specifies the relocatable name of the common block.
Expression	Specifies the absolute expression which gives length of common block *Name* in bytes.
Number	Specifies the optional alignment of common block *Name*.

Example

```
        .comm proc,5120

# proc is an uninitialized common block of
# storage 5120 bytes long which is
# globally visible.
# Assembler SourceFile A contains:
        .comm st,1024
# Assembler SourceFile B contains:
        .globl st[RW]
        .csect st[RW]
        .long 1
        .long 2

# Using st in the above two programs refers to
# Control Section st in Assembler SourceFile B.
```

Implementation Specifics

This pseudo–op is part of Application Development Toolkit in AIX Base.

Related Information

The **.align** pseudo–op, **.csect** pseudo–op, **.globl** pseudo–op, **.lcomm** pseudo–op, **.long** pseudo–op.

Pseudo-Ops Overview on page 6–2.

.csect Pseudo–op

Purpose

Groups code and/or data into a Control Section and gives that Control Section a name, a storage class, and an alignment.

Syntax

.csect *Qualname*[,*Number*]

where *Qualname* = [*Name*][[*StorageClass*]]

Note: The bold–faced brackets containing *StorageClass* are part of the syntax and **do not** specifiy optional parameters.

Description

The **.csect** pseudo–op groups code and/or data into a Control Section and gives that Control Section a name, a storage class, and an alignment.

- If no *Name* is specified in the **.csect** pseudo–op, then the Control Section is unnamed.
- If no *StorageClass* is specified in the **.csect** pseudo–op, then the [PR] *StorageClass* is the default.

Each Control Section has a storage class associated with it that is specified in the qualification part of *Qualname*. The storage class determines the object data section, specifically the **.text**, **.data**, or **.bss** section, in which the Control Section is grouped. The **.text** section contains read only data. The **.data** and **.bss** sections contain read write data.

The storage class also indicates what kind of data should be contained within the Control Section. Many of the storage classes listed have specific implementation and convention details. In general, instructions can be contained within csects of storage class PR. Modifiable data can be contained within csects of storage class RW.

A csect Control Section is of one of the following storage classes:

.text Section Storage Classes

PR Program Code
Identifies the sections that provide executable instructions for the module.

RO Read Only Data
Identifies the sections that contain constants that are not modified during execution.

DB Debug Table
Identifies a class of sections that has the same characteristics as read only data.

GL Glue Code
Identifies a section that has the same characteristics as Program Code. This type of section has code to interface with a routine in another module. Part of the interface code requirement is to maintain TOC addressability across the call.

XO Extended Op
Identifies a section of code that is to be treated as a pseudo machine instruction.

SV SVC
Identifies a section of code that is to be treated as a supervisor call.

TB Traceback Table
Identifies a section that contains data associated with a traceback table.

TI Traceback Index
Identifies a section that contains data associated with a traceback index.

.data Section Storage Classes

TC0 TOC Anchor used only by the predefined TOC symbol
Identified the special symbol TOC. Used only for the TOC anchor.

TC TOC Entry
Identifies data that will reside in the TOC.

UA Unknown Type
Identifies a section that contains data of an unknown storage class.

RW Read Write Data
Identifies a section that contains data that is known to require change during execution.

DS Descriptor
Identifies a function descriptor. This information is used to describe function pointers in languages such as C and FORTRAN.

.bss Section Storage Classes

BS BSS class
Identifies a section that contains uninitialized read write data.

UC Unnamed FORTRAN Common
Identifies a section that contains read write data.

- All of the Control Sections with the same *Qualname* are grouped together, and a section can be continued with a **.csect** statement with the same *Qualname*. Two Control Sections can have the same *Name* and different classes.
- A Control Section is relocated as a body.
- Control Sections with no *Name* are identified with their class, and there can be an unnamed Control Section of each class. They are specified with a *Qualname* that only has a storage class (for instance, **.csect** [RW] has the *Qualname* [RW]).
- If no **.csect** pseudo-op is specified before any instructions appear, then an unnamed Program Code [PR] Control Section is assumed.
- You cannot use **.csect** to define a Control Section of type CM (Common). Control Sections defined with the **.csect** Pseudo–op are of type SD (Section Definition). To define type CM Control Sections, you must use the **.comm** and **.lcomm** pseudo-ops.
- *Number* is specified as the log base 2 of the desired alignment. For example, an alignment of 8 (or doubleword) would be 3 and an alignment of 2048 would be 11. This is similar to the argument for the **.align** pseudo-op. Alignment occurs at the beginning of the **.csect**. Each element of the **.csect** is not individually aligned.

- Do not label **.csect** statements. The **.csect** may be referred to as its *Qualname*, and labels may be placed on individual elements of the **.csect**.

Parameters

Number Specifies an expression that evaluates to an integer from 0 to 31 inclusive.

Qualname Specifies a *Name* and *StorageClass* for the Control Section. If *Name* not given, the csect is identified with its *StorageClass*. If *StorageClass* not given, the csect defaults to storage class [PR].

Example

```
# A csect of name proga with Program Code Storage Class.
.csect proga[PR]
lh      30,0x64(5)
# A csect of name pdata_ with Read Only Storage Class.
.csect pdata_[RO]

l1:     .long  0x7782
l2:     .byte  'a,'b,'c,'d,'e
.csect  [RW],3  # An unnamed csect with Read Write

                        # Storage Class and doubleword
                        # alignment.

.float -5
```

Implementation Specifics

This pseudo–op is part of Application Development Toolkit in AIX Base.

Related Information

The **.comm** pseudo–op, **.globl** pseudo–op, **.lcomm** pseudo–op.

Pseudo-Ops Overview on page 6–2.

.double Pseudo–op

Purpose

Stores a double floating–point constant at the next fullword location.

Syntax

.double *FloatingConstant*

Description

The **.double** pseudo-op stores a double floating–point constant at the next fullword location. Fullword alignment occurs if necessary.

Parameters

FloatingConstant Specifies the double floating–point constant to be assembled.

Example

```
.double 3.4
.double –77
.double 134E12
.double 5e300
.double 0.45
```

Implementation Specifics

This pseudo–op is part of Application Development Toolkit in AIX Base.

Related Information

The **.float** pseudo–op.

Pseudo-Ops Overview on page 6–2.

.drop Pseudo–op

Purpose

Stops using a specified register as a base register.

Syntax

.drop *Number*

Description

The **.drop** pseudo-op stops a program from using register *Number* as a base register in operations. The **.drop** pseudo-op does not have to precede the **.using** pseudo-op when changing the base address, and the **.drop** pseudo-op does not have to appear at the end of a program.

Parameters

Number Specifies an expression that evaluates to an integer from 0 to 31 inclusive.

Example

```
.using _subrA,5

        # Register 5 can now be used for addressing
        # with displacements calculated
        # relative to _subrA.

        # .using does not load GPR 5 with the address
        # of _subrA. The program must contain the
        # appropriate code to ensure this at runtime.

   .
   .
   .
.drop 5

        # Stop using Register 5.
.using _subrB,5
        # Now the assembler calculates
        # displacements relative to _subrB
```

Implementation Specifics

This pseudo–op is part of Application Development Toolkit in AIX Base.

Related Information

The **.using** Pseudo–op.

Pseudo-Ops Overview on page 6–2.

.dsect Pseudo–op

Purpose

Identifies the beginning or the continuation of a dummy control section.

Syntax

.dsect *Name*

Description

The **.desct** pseudo-op identifies the beginning or the continuation of a dummy control section. Actual data declared in a dummy control section is ignored; only the location counter is incremented. All labels in a dummy section are considered to be offsets relative to the beginning of the dummy section. A **.dsect** that has the same name as a previous **.dsect** is a continuation of that dummy section.

The **.dsect** pseudo–op can declare a data template which can then be used to map out a block of storage. The **.dsect** pseudo-op can also be used with the **.using** pseudo-op.

Example

```
        .dsect datal
                # 1 Fullword
d1:     .long 0
                # 10 Halfwords
d2:     .short 0,0,0,0,0,0,0,0,0,0
                # 15 bytes
d3:     .byte 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
        .align 3                    #Align to a double word.
d4:     .space 64                   #Space 64 bytes
.csect main[PR]
.using datal,7
l 5,d2
# This will actually load
# the contents of the
# effective address calculated
# by adding the offset d2 to
# that in GPR 7 into GPR 5.
```

Implementation Specifics

This pseudo–op is part of Application Development Toolkit in AIX Base.

Related Information

The **.csect** pseudo–op, **.using** pseudo–op.

Pseudo-Ops Overview on page 6–2.

.eb Pseudo–op

Purpose

Identifies the end of an inner block and provides additional information specific to the end of an inner block.

Syntax

.eb *Number*

Description

The **.eb** pseudo–op identifies the end of an inner block and provides symbol table information necessary for the use of the symbolic debugger.

This pseudo–op has no other effect on assembly and is customarily inserted by a compiler.

Parameters

Number Specifies a line number in the original source file on which the inner block ends.

Example

```
.eb     10
```

Implementation Specifics

This pseudo–op is part of Application Development Toolkit in AIX Base.

Related Information

The **.bb** pseudo–op.

Pseudo-Ops Overview on page 6–2.

.ec Pseudo–op

Purpose

Identifies the end of a common block and provides additional information specific to the end of a common block.

Syntax

.ec

Description

The **.ec** pseudo–op identifies the end of a common block and provides symbol table information necessary for the use of the symbolic debugger.

This pseudo–op has no other effect on assembly and is customarily inserted by a compiler.

Example

```
.bc     "commonblock"
.ec
```

Implementation Specifics

This pseudo–op is part of Application Development Toolkit in AIX Base.

Related Information

The **.bc** pseudo–op.

Pseudo-Ops Overview on page 6–2.

.ef Pseudo–op

Purpose

Identifies the end of a function and provides additional information specific to the end of a function.

Syntax

.ef *Number*

Description

The **.ef** pseudo–op identifies the end of a function and provides symbol table information necessary for the use of the symbolic debugger.

This pseudo–op has no other effect on assembly and is customarily inserted by a compiler.

Parameters

Number Specifies a line number in the original source file on which the function ends.

Example

```
.ef     10
```

Implementation Specifics

This pseudo–op is part of Application Development Toolkit in AIX Base.

Related Information

The **.bf** pseudo–op.

Pseudo-Ops Overview on page 6–2.

.ei Pseudo–op

Purpose

Identifies the end of an included file and provides additional information specific to the end of an included file.

Syntax

.ei

Description

The **.ei** pseudo–op identifies the end of an included file and provides symbol table information necessary for the use of the symbolic debugger.

This pseudo–op has no other effect on assembly and is customarily inserted by a compiler.

Example

```
.ei     "file.s"
```

Implementation Specifics

This pseudo–op is part of Application Development Toolkit in AIX Base.

Related Information

The **.bi** pseudo–op.

Pseudo-Ops Overview on page 6–2.

.es Pseudo–op

Purpose

Identifies the end of a static block and provides additional information specific to the end of a static block.

Syntax

.es

Description

The **.es** pseudo–op identifies the end of a static block and provides symbol table information necessary for the use of the symbolic debugger.

This pseudo–op has no other effect on assembly and is customarily inserted by a compiler.

Example

```
.lcomm cgdat, 0x2b4
.csect .text[PR]
.bs cgdat

        .stabx "ONE:1=Ci2,0,4;",0x254,133,0
        .stabx "TWO:S2=G5TWO1:3=Cc5,0,5;,0,40;;",0x258,133,8
.es
```

Implementation Specifics

This pseudo–op is part of Application Development Toolkit in AIX Base.

Related Information

The **.bs** pseudo–op.

Pseudo-Ops Overview on page 6–2.

.extern Pseudo–op

Purpose

Identifies a symbol defined in another source module.

Syntax

.extern *Name*

Description

The **.extern** instruction identifies *Name* as a symbol defined in another source module, and *Name* becomes an external symbol. Any external symbols used in the current assembly that are not defined in the current assembly must be declared with an **.extern** statement. If a symbol defined locally appears in an **.extern** statement, then it is like using that symbol in a **.globl** statement. If a symbol not defined locally appears in a **.globl** statement, then it is like using that symbol in and **.extern** statement. An undefined symbol will be flagged as an error.

Parameters

Name Specifies an operand which is an external symbol and can be a *Qualname*.

Qualname Specifies a *Name* and *StorageClass* for the Control Section.

Example

```
        .extern proga[PR]
        .toc
T.proga:        .tc proga[TC],proga[PR]
```

Implementation Specifics

This pseudo–op is part of Application Development Toolkit in AIX Base.

Related Information

The **.csect** pseudo–op, **.globl** pseudo–op.

Pseudo-Ops Overview on page 6–2.

.file Pseudo–op

Purpose

Identifies a source file name.

Syntax

.file *StringConstant*

Description

The **.file** psuedo–op provides symbol table information necessary for the use of the symbolic debugger and linkage editor.

This pseudo–op is customarily inserted by a compiler and has no other effect on assembly.

Parameters

StringConstant Specifies the file name of the original source file.

Example

```
.file "main.c"
```

Implementation Specifics

This pseudo–op is part of Application Development Toolkit in AIX Base.

Related Information

The **.function** pseudo–op.

Pseudo-Ops Overview on page 6–2.

.float Pseudo–op

Purpose

Stores a floating–point constant at the next fullword location.

Syntax

.float *FloatingConstant*

Description

The **.float** stores a floating–point constant at the next fullword location. Fullword alignment occurs if necessary.

Parameters

FloatingConstant Specifies floating point constant to be assembled.

Example

```
.float 3.4
.float –77
.float 134E–12
```

Implementation Specifics

This pseudo–op is part of Application Development Toolkit in AIX Base.

Related Information

The **.double** pseudo–op.

Pseudo-Ops Overview on page 6–2.

.function Pseudo–op

Purpose

Identifies a function and provides additional information specific to the function.

Syntax

.function *Name,Expression1,Expression2,Expression3*

Description

The **.function** pseudo–op identifies a function and provides symbol table information necessary for the use of the symbolic debugger.

This pseudo–op is customarily inserted by a compiler and has no other effect on assembly.

Parameters

Name	Represents the function *Name* and should be defined as a symbol or control section *Qualname* in the current assembly.
Expression1	Represents the top of the function.
Expression2	Represents the storage class of the function.
Expression3	Represents the type of the function.
Qualname	Specifies a *Name* and *StorageClass* for the Control Segment.

Example

```
.globl  .hello[pr]
.csect  .hello[pr]
.function       .hello[pr],L.1B,16,044
L.1B:
```

Implementation Specifics

This pseudo–op is part of Application Development Toolkit in AIX Base.

Related Information

The **.bf** pseudo–op, **.ef** pseudo–op, **.file** pseudo–op.

Pseudo-Ops Overview on page 6–2.

.globl Pseudo–op

Purpose

Makes a symbol globally visible to the linker.

.globl *Name*

Description

The **.globl** pseudo-op makes the symbol *Name* globally visible to the linker and available to any file that is linked to the file in which the **.globl** pseudo–op occurs.

- If the **.globl** pseudo–op is not used for a symbol, then that symbol is, unless otherwise effected, only visible within the current assembly, and not to other modules that may later be linked to the current assembly. Alternately, the **.extern** pseudo–op can be used to effect visibility.
- If *Name* is defined in the current assembly, its type and value arise from that definition, not the **.globl** definition.
- The binder maps all common segments with the same name into the same memory. If the name is declared **.globl** and defined in one of the segments, this has the same effect as declaring the common symbols to be **.globl** in all segments. In this way, common memory can be initialized.

Parameters

Name Represents any label or symbol that is defined locally and requires external visibility.

Example

```
        .globl main
main:
        .csect data[rw]
        .globl data[rw]
```

Implementation Specifics

This pseudo–op is part of Application Development Toolkit in AIX Base.

Related Information

The **.comm** pseudo–op, **.extern** pseudo–op.

Pseudo-Ops Overview on page 6–2.

.hash Pseudo–op

Purpose

Associates a hash value with an external symbol.

Syntax

.hash *Name,StringConstant*

Description

The **.hash** pseudo-op associates a hash value with an external symbol. Hash values are generated by compilers of strongly typed languages. The hash code for a symbol can only be set once in an assembly.

Parameters

Name Represents a symbol. Because this should be an external symbol, *Name* should appear in an **.extern** or **.global** statement.

StringConstant Represents characters that represent a hexadecimal hash code and must be in the set [0–9A–F] or [0–9a–F].

Example

```
.hash a[pr],"ff0a2cc12365de30"
```

Implementation Specifics

This pseudo–op is part of Application Development Toolkit in AIX Base.

Related Information

The **.extern** pseudo–op, **.globl** pseudo–op.

Pseudo-Ops Overview on page 6–2.

.lcomm Pseudo–op

Purpose

Defines a local uninitialized block of storage.

Syntax

.lcomm *Name1,Expression*[,*Name2*]

Description

The **.lcomm** instruction defines a local uninitialized block of storage called a local common (LC) section. At runtime, this storage block will be reserved when the local common (LC) section is allocated at the end of the **.data** section. This storage block is for uninitialized data.

Use **.lcomm** with local uninitialized data. This means that the data will probably not be accessed outside the local assembly.

- The symbol *Name1* is a label at the top of the local uninitialized block of storage. The location counter for this local common (LC) section is incremented by *Expression.* A specific local common (LC) section can be specified by the *Name2* operand. Otherwise an unnamed section is used.

Parameters

Name1 Represents a relocatable symbol. The symbol *Name1* is a label at the top of the local uninitialized block of storage. *Name1* does not appear in the symbol table unless it is the operand of a **.globl** statement.

Expression Represents an absolute expression that is defined in the first pass of the assembler. The *Expression* expression also increments the location counter for the local common (LC) section.

Name2 Represents a Control Section name that has storage class **BS** and storage type **CM**. The *Name2* operand allows the programmer to specify the **BS** Control Section for the allocated storage. If a specific local common (LC) section is not specified by the *Name2* operand, an unnamed section is used.

Example

```
.lcomm buffer,5120
        # Can refer to this 5K
        # of storage as "buffer".


.lcomm b3,4,proga
        # b3 will be a label in a csect of class BS
        # and type CM with name "proga".
```

Implementation Specifics

This pseudo–op is part of Application Development Toolkit in AIX Base.

Related Information

The **.comm** pseudo–op.

Pseudo-Ops Overview on page 6–2.

.line Pseudo–op

Purpose

Identifies a line number and provides additional information specific to the line number.

Syntax

.line *Number*

Description

The **.line** pseudo–op identifies a line number and provides symbol table information necessary for the use of the symbolic debugger.

This pseudo–op is customarily inserted by a compiler and has no other effect on assembly.

Parameters

Number Represents a line number of the original source file.

Example

```
.globl .hello[pr]
.csect .hello[pr]
.align 1
.function       .hello[pr],L.1B,16,044
.stabx "hello:f–1",0,142,0
.bf 2
.line 1
.line 2
```

Implementation Specifics

This pseudo–op is part of Application Development Toolkit in AIX Base.

Related Information

The **.bf** pseudo–op, **.function** pseudo–op.

Pseudo-Ops Overview on page 6–2.

.long Pseudo–op

Purpose

Assembles expressions into consecutive fullwords.

Syntax

.long *Expression*[,*Expression*,...]

Description

The **.long** pseudo-op assembles expressions into consecutive fullwords. Fullword alignment occurs as necessary.

Parameters

Expression Represents any expression to be assembled into fullwords.

Example

```
.long 24,3,fooble–333,0
```

Implementation Specifics

This pseudo–op is part of Application Development Toolkit in AIX Base.

Related Information

The **.byte** pseudo–op, **.short** pseudo–op, **.vbyte** pseudo–op.

Pseudo-Ops Overview on page 6–2.

.org Pseudo–op

Purpose

Sets the value of the current location counter.

Syntax

.org *Expression*

Description

The **.org** pseudo-op sets the value of the current location counter to *Expression*. This pseudo–op can also decrement a location counter. The assembler is Control Section oriented; therefore, absolute expressions or expressions which cause the location counter to go outside of the current Control Section are not allowed.

Parameters

Expression Represents the value of the current location counter.

Example

```
# Assume assembler location counter is 0x114.
.org $+100
#Skip 100 decimal byte (0x64 bytes).
.
.
# Assembler location counter is now 0x178.
```

Implementation Specifics

This pseudo–op is part of Application Development Toolkit in AIX Base.

Related Information

The **.space** pseudo–op.

Pseudo-Ops Overview on page 6–2.

.rename Pseudo–op

Purpose

Creates a synonym or alias for an illegal or undesirable name.

Syntax

.rename *Name,StringConstant*

Description

The **.rename** pseudo-op changes *Name* to *StringConstant* for all external references at the end of assembly. Internal references to the local assembly are made to *Name*. The externally visible *Name* is *StringConstant*. The **.rename** pseudo–op is useful in referencing symbol names that are otherwise illegal in the assembler syntax.

Parameters

Name	Represents a symbol. To be externally visible, *Name* must appear in an **.extern** or **.globl** statement.
StringConstant	Represents value which*Name* is changed to at end of assembly.

Example

```
.rename toc_of_er,"#ER"
.rename a[pr],"$PR"
```

Implementation Specifics

This pseudo–op is part of Application Development Toolkit in AIX Base.

Related Information

The **.extern** pseudo–op, **.globl** pseudo–op.

Pseudo-Ops Overview on page 6–2.

.set Pseudo–op

Purpose

Sets a symbol equal to an expression in both type and value.

Syntax

.set *Name,Expression*

Description

The **.set** pseudo-op sets the symbol *Name* equal to the expression *Expression* in type and in value. Using **.set** may help to avoid errors with a frequently used expression. Equate the expression to a symbol, then refer to the symbol rather than the expression. To change the value of the expression, only change it within the **.set** statement. However, reassembling the program is necessary since **.set** assignments occur only at assembly time.

The expression *Expression* can only refer to symbols within the same Control Section. The symbols do not have to be within the Control Section where the **.set** appears, only in the same Control Section as other symbols in the expression.

Parameters

Name Represents a symbol which may be used before its definition in a **.set** statement; forward references are allowed within a module.

Expression Refers to symbols within the same Control Section. The symbols do not have to be within the Control Section where the **.set** appears, only in the same Control Section as other symbols in the expression. It can also refer to a register number but not the contents of the register at runtime. The *Expression* parameter cannot be an undefined external expression.

Example

```
.set ap,14 # Assembler assigns value 14
           # to the symbol ap — ap
           # is absolute.


lil ap,2

        # Assembler substitutes value 14
        # for the symbol.
        # Note that ap is a register
        # number in context
        # as lil's operand.
```

Implementation Specifics

This pseudo–op is part of Application Development Toolkit in AIX Base.

Related Information

Understanding Expressions on page 2–12.

Pseudo-Ops Overview on page 6–2.

.short Pseudo–op

Purpose

Assembles expressions into consecutive halfwords.

Syntax

.short *Expression*[,*Expression*,...]

Description

The **.short** pseudo-op assembles *Expressions* into consecutive halfwords. Halfword alignment occurs as necessary.

Parameters

Expression Represents expressions which the instruction assembles into halfwords. *Expression* cannot refer to the contents of any register, and if *Expression* is longer than a halfword, it is truncated on the left.

Example

```
.short 1,0x4444,fooble–333,0
```

Implementation Specifics

This pseudo–op is part of Application Development Toolkit in AIX Base.

Related Information

The **.byte** pseudo–op, **.long** pseudo–op, **.vbyte** pseudo–op.

Pseudo-Ops Overview on page 6–2.

.space Pseudo–op

Purpose

Skips a specified number of bytes in the output file and fills them with binary zeros.

Syntax

.space *Number*

Description

The **.space** skips a number of bytes, specified by *Number*, in the output file and fills them with binary zeros. The **.space** pseudo-op may be used to reserve a chunk of storage in a Control Section.

Parameters

Number Represents an absolute expression that specifies the number of bytes to skip.

Example

```
.csect data[rw]
.space 444
  .
  .
foo:    # foo currently located at offset 0x1BC within
        # csect data[rw].
```

Implementation Specifics

This pseudo–op is part of Application Development Toolkit in AIX Base.

Related Information

Pseudo-Ops Overview on page 6–2.

.stabx Pseudo–op

Purpose

Provides additional information required by the debugger.

Syntax

.stabx *StringConstant,Expression1,Expression2,Expression3*

Description

The **.stabx** pseudo–op provides additional information required by the debugger. The assembler places the *StringConstant* argument, which provides required Stabstring information for the debugger, in the .debug section.

This pseudo–op is customarily inserted by a compiler.

Parameters

StringConstant	Provides required Stabstring information to the debugger.
Expression1	Represents the symbol value of the character string.
Expression2	Represents the storage class of the character string.
Expression3	Represents the symbol type of the character string.

Example

```
.stabx "INTEGER:t2=–1",0,140,4
```

Implementation Specifics

This pseudo–op is part of Application Development Toolkit in AIX Base.

Related Information

The **.function** pseudo–op.

Pseudo-Ops Overview on page 6–2.

See "dbx Stabstring Grammar (C, COBOL, Pascal, FORTRAN, and Modula–2)" in the a.out File Format article in *Files Reference*.

.string Pseudo–op

Purpose

Assembles character values into consecutive bytes and terminates the string with a null character.

Syntax

.string *StringConstant*

Description

The **.string** pseudo-op assembles the character values represented by *StringConstant* into consecutive bytes and terminates the string with a null character.

Parameters

StringConstant Represents a string of character values assembled into consecutive bytes.

Example

```
mine:   .string "Hello, world!"
# This produces
# 0x48656C6C6F2C20776F726C642100.
```

Implementation Specifics

This pseudo–op is part of Application Development Toolkit in AIX Base.

Related Information

The **.byte** pseudo–op, **.vbyte** pseudo–op.

Pseudo-Ops Overview on page 6–2.

.tbtag Pseudo–op

Purpose

Defines a debug traceback tag that can be used to perform tracebacks when debugging programs.

Syntax

.tbtag *Expression1,Expression2,Expression3,Expression4,Expression5,Expression6,Expression7,Expression8*[*,Expression9,Expression10,Expression11,Expression12,Expression13,Expression14,Expression15,Expression16*]

Description

The **.tbtag** pseudo–op defines a traceback tag by assembling the *Expressions* into consecutive bytes, words, and halfwords, depending on field requirements. Traceback information is customarily inserted by a compiler.

Parameters

Expression1	FORMAT_TYPE	Byte
Expression2	LANG_IDENT	Byte
Expression3	IS_GL,IS_EPROL,SHORT_TB, INT_PROC,HAS_CTL, MILI_CODE,FP_PRESENT, LOG_ABORT	Byte
Expression4	INT_HANDLER,NAME_PRESENT, SPARE,CL_DIS_INV,SAVES_CR, SAVES_LR	Byte
Expression4	INT_HANDLER,NAME_PRESENT, SPARE,CL_DIS_INV,SAVES_CR, SAVES_LR	Byte
Expression5	STORES_BC,SPARE,FP_SAVED	Byte
Expression6	SPARE,SPARE,GP_SAVED	Byte
Expression7	PARMCNTFX	Byte
Expression8	PARMCNTFL,PARMCNTV	Byte
Expression9	PARMTYPE	Word
Expression10	CODELEN	Word
Expression11	HAND_MASK	Word
Expression12	CTL_INFO	Word
Expression13	CTL_INFO_DISP	Word
Expression14	NAME_LENGTH	Halfword
Expression15	NAME	Byte

Expression16	ALLOCA_REG	Byte

Example

```
.tbtag 1,0,0xff,0,0,16
```

Implementation Specifics

This pseudo–op is part of Application Development Toolkit in AIX Base.

Related Information

The **.byte** pseudo–op.

Pseudo-Ops Overview on page 6–2.

.tc Pseudo–op

Purpose

Assembles expressions into a TOC entry.

Syntax

.tc [*Name*][TC],*Expression*[,*Expression*,...]

Note: The bold–faced brackets containing TC are part of the syntax and **do not** specifiy optional parameters.

Description

The **.tc** pseudo-op assembles *Expressions* into a TOC entry. A **.tc** statement can only appear inside the scope of a **.toc** pseudo-op. A TOC entry can be relocated as a body. TOC entry statements can have local labels, which will be relative to the beginning of the entire TOC as declared by the first **.toc** statement. Addresses contained in the TOC entry can be accessed using these local labels and the TOC Register GPR 2.

TOC entries that contain only one address are subject to being combined by the binder. This can occur if the TOC entries have the same name and reference the same csect (symbol). Be careful when coding TOC entries that reference non–zero offsets within a csect. To prevent unintended combining of TOC entries, unique names should be assigned to TOC entries that reference different offsets within a csect.

Parameters

Name Specifies name of the TOC entry created. The *StorageClass* is TC for TOC entires. *Name*[TC] can be used to refer to the TOC entry where appropriate.

Expression Specifies symbol or expression which goes into TOC entry.

Example

```
.toc
# Create three TOC entries, the first
# with the name proga, the second
# with the name progb, and the last
# unnamed.

T.proga:        .tc proga[TC],progr[RW],dataA
T.progb:        .tc progb[TC],proga[PR],progb[PR]
T.progax:       .tc  proga[TC],dataB
                .tc       [TC],dataB
                .csect proga[PR]

# A .csect should precede any statements following a
# .toc/.tc section which do not belong in the TOC.
                l 5,T.proga(2)  # The address of progr[RW]
                                # is loaded into GPR 5.

                l 5,T.progax(2) # The address of progr[RW]
                                # is loaded into GPR 5.
                l 5,T.progb+4(2)        # The address of progb[PR]
                                # is loaded into GPR 5.
```

Implementation Specifics

This pseudo–op is part of Application Development Toolkit in AIX Base.

Related Information

The **.csect** pseudo–op, .toc **pseudo–op**, **.tocof** pseudo–op.

Pseudo-Ops Overview on page 6–2.

.toc Pseudo–op

Purpose

Defines the table of contents of a module.

Syntax

.toc

Description

The **.toc** pseudo–op defines the table of contents (TOC) anchor of a module. Entries in the TOC section can be declared with **.tc** pseudo-op within the scope of the **.toc** pseudo–op. The **.toc** pseudo–op has scope similar to that of a **.csect** pseudo-op. The TOC can be continued throughout the assembly wherever a **.toc** appears.

Example

```
.toc
# Create two TOC entries. The first
# entry, named proga, is of type TC
# and contains the address of
# proga[RW] and dataA.

# The second entry, named progb, is of type TC
# and contains the address of
# progb[PR] and progc[PR].
T.proga:        .tc proga[TC],proga[RW],dataA
T.progb:        .tc progb[TC],progb[PR],progc[PR]

.csect proga[RW]
# A .csect should precede any statements following a
# .toc/.tc section which do not belong in the TOC.
.long TOC[tc0]
# The address of the TOC for this module is placed in
# a fullword.
```

Implementation Specifics

This instruction is part of Application Development Toolkit in AIX Base.

Related Information

The **.tc** pseudo–op, **.tocof** pseudo–op.

Pseudo-Ops Overview on page 6–2.

.tocof Pseudo–op

Purpose

Allows for the definition of a local symbol as the table of contents of an external symbol so that the local symbol can be used in expressions.

Syntax

.tocof *Name1,Name2*

Description

The **.tocof** pseudo-op makes *Name2* globally visible to the linker and marks *Name1* as the table of contents (TOC) of another module that contains the symbol *Name2*. This allows the definition of a local symbol as the TOC of an external symbol so that the local symbol can be used in expressions or to refer to the TOC of another module, usually in a **.tc** statement. This pseudo-op generates a Relocation Dictionary entry (RLD) that causes this data to be initialized to the address of the TOC external symbols. The **.tocof** pseudo–op can be used for inter-module calls that require the caller to first load up the address of the called module's TOC before transferring control.

Parameters

Name1 Specifies a local symbol that acts as the TOC of a module that contains *Name2*. The symbol *Name1* should appear in **.tc** statements.

Name2 Specifies an external symbol that exists within a modual that contains a TOC.

Example

```
tocbeg: .toc
apb:    .tc [tc],pb,tpb
# This is an unnamed TOC entry
# that contains two addresses:
# the address of pb and
# the address of the TOC
# containing pb.
.tocof tpb,pb
.set always,0x14
.csect [PR]
.using tocbeg,rtoc
l 14,apb
# Load R14 with the address
# of pb.
l rtoc,apb+4
# Load the TOC register with the
# address pb's TOC.
mtspr lr,14
# Move to Link Register.
bcr always,0
# Branch Conditional Register branch
# address is contained in the Link
# register.
```

Implementation Specifics

This instruction is part of Application Development Toolkit in AIX Base.

Related Information

The **.tc** pseudo–op, **.toc** pseudo–op.

Pseudo-Ops Overview on page 6–2.

.using Pseudo–op

Purpose

Assigns a base register number.

Syntax

.using *Expression,Register*

Description

The **.using** pseudo-op bases relocatable expressions from *Register*, assuming that *Register* contains the relocatable program address of *Expression* at runtime.

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 the base *Register* 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).

The **.using** pseudo–op can be issued on the csect name and all the labels in the csect are referenced via the using register. Other types of external symbols are not allowed (**.extern**).

Parameters

Register Represents the register number for relocatable expressions. It must be absolute and must evaluate to an integer from 0 to 31 inclusive.

Expression Specifies a label or an expression involving a label that represents the displacement or relative offset into the program. It must be relocatable but cannot be an absolute symbol. The *Expression* parameter can be an external symbol if the symbol is a csect or TOC entry defined within the assembly.

Example

```
.csect data[rw]
.long 0x0,0x0
dl:     .long 0x25
# A read/write csect contains the label dl.
.csect text[pr]
.using data[rw],12
l 4,dl
# This will actually load the contents of
# the effective address, calculated by
# adding the address dl to the address in
# GPR 12, into GPR 4.
```

Implementation Specifics

This pseudo–op is part of Application Development Toolkit in AIX Base.

Related Information

The **.csect** pseudo–op, **.drop** pseudo–op.

Pseudo-Ops Overview on page 6–2.

.vbyte Pseudo–op

Purpose

Assembles the value represented by an expression into consecutive bytes.

Syntax

.vbyte *Number,Expression*

Description

The **.vbyte** pseudo–op assembles the value represented by the *Expression* into consecutive *Number* bytes.

Parameters

Number Specifies a number of consecutive bytes. *Number* must range between 1 and 4.

Expression Specifies a value that is assembled into consecutive bytes. The *Expression* parameter cannot contain externally defined symbols, and if *Expression* is longer than *Number* bytes, it will be truncated on the left.

Example

```
.csect data[RW]
mine:   .vbyte 3,0x37CCFF
# This pseudo–op also accepts character constants.
.vbyte 1,'c

# Load GPR 4 with address of .csect data[RW].
.csect text[PR]
l 3,mine(4)
# GPR 3 now holds 0x37CCFF.
```

Implementation Specifics

This pseudo–op is part of Application Development Toolkit in AIX Base.

Related Information

The **.byte** pseudo–op.

Pseudo-Ops Overview on page 6–2.

.xline Pseudo–op

Purpose

Represents a line number.

Syntax

.xline *Number1,StringConstant*[,*Number2*]

Description

The **.xline** pseudo–op provides additional file and line number information to the assembler. *Number2* can be used to generate **.bi** and **.ei** type entries for use by symbolic debuggers. This pseudo–op is customarily inserted by the M4 macroprocessor.

Parameters

Number1	Represents the line number of the original source file.
StringConstant	Represents the filename of the original source file.
Number2	Represents the C_BINCL and C_EINCL classes that indicate the beginning and ending of an included file.

Example

```
.xline 1,"hello.c",108
.xline 2,"hello.c"
```

Implementation Specifics

This pseudo–op is part of Application Development Toolkit in AIX Base.

Related Information

Pseudo-Ops Overview on page 6–2.

Appendix A. Opcode and Mnemonic Tables

Instruction Set, Indexed by Primary Opcode

Mnemonic	Instruction	Format	Primary Opcode	Extended Opcode
ti	Trap Immediate	D	03	
muli	Multiply Immediate	D	07	
sfi	Subtract From Immediate	D	08	
dozi	Difference Or Zero Immediate	D	09	
cmpli	Compare Logical Immediate	D	10	
cmpi	Compare Immediate	D	11	
ai	Add Immediate	D	12	
ai.	Add Immediate And Record	D	13	
cal	Compute Address Lower	D	14	
cau	Compute Address Upper	D	15	
bc[l][a]	Branch Conditional	B	16	
svc[l][a]	Supervisor Call	SC	17	
b[l][a]	Branch	I	18	
crand	Condition Register AND	XL	19	257
crandc	Condition Register AND With Complement	XL	19	129
creqv	Condition Register Equivalent	XL	19	289
crnand	Condition Register NAND	XL	19	225
crnor	Condition Register NOR	XL	19	33
cror	Condition Register OR	XL	19	449
crorc	Condition Register OR With Complement	XL	19	417
crxor	Condition Register XOR	XL	19	193
bcc[l]	Branch Conditional To Count Register	XL	19	528
bcr[l]	Branch Conditional Register	XL	19	16
mcrf	Move Condition Register Field	XL	19	0
rlimi[.]	Rotate Left Immediate Then Mask Insert	M	20	
rlinm[.]	Rotate Left Immediate Then AND With Mask	M	21	
rlmi[.]	Rotate Left Then Mask Insert	M	22	
rlnm[.]	Rotate Left Then AND With Mask	M	23	
oril	OR Immediate Lower	D	24	
oriu	OR Immediate Upper	D	25	
xoril	XOR Immediate Lower	D	26	
xoriu	XOR Immediate Upper	D	27	

Mnemonic	Instruction	Format	Primary Opcode	Extended Opcode
andil.	AND Immediate Lower	D	28	
andiu.	AND Immediate Upper	D	29	
a[o][.]	Add	XO	31	10
abs[o][.]	Absolute	XO	31	360
ae[o][.]	Add Extended	XO	31	138
ame[o][.]	Add To Minus One Extended	XO	31	234
and[.]	AND	X	31	28
andc[.]	AND With Complement	X	31	60
aze[o][.]	Add To Zero Extended	XO	31	202
cax[o][.]	Compute Address	XO	31	266
cmp	Compare	X	31	0
cmpl	Compare Logical	X	31	32
cntlz[.]	Count Leading Zeroes	X	31	26
div[o][.]	Divide	XO	31	331
divs[o][.]	Divide Short	XO	31	363
doz[o][.]	Difference Or Zero	XO	31	264
eqv[.]	Equivalent	X	31	284
exts[.]	Extend Sign	X	31	922
lbrx	Load Byte Reverse Indexed	X	31	534
lbzux	Load Byte And Zero With Update Indexed	X	31	119
lbzx	Load Byte And Zero Indexed	X	31	87
lfdux	Load Floating–Point Double With Update Indexed	X	31	631
lfdx	Load Floating–Point Double Indexed	X	31	599
lfsux	Load Floating–Point Single With Update Indexed	X	31	567
lfsx	Load Floating–Point Single Indexed	X	31	535
lhaux	Load Half Algebraic With Update Indexed	X	31	375
lhax	Load Half Algebraic Indexed	X	31	343
lhbrx	Load Half Byte Reverse Indexed	X	31	790
lhzux	Load Half And Zero With Update Indexed	X	31	311
lhzx	Load Half And Zero Indexed	X	31	279
lscbx[.]	Load String And Compare Byte Indexed	X	31	277
lsi	Load String Immediate	X	31	597

Mnemonic	Instruction	Format	Primary Opcode	Extended Opcode
lsx	Load String Indexed	X	31	533
lux	Load With Update Indexed	X	31	55
lx	Load Indexed	X	31	23
maskg[.]	Mask Generate	X	31	29
maskir[.]	Mask Insert From Register	X	31	541
mcrxr	Move To Condition Register From XER	X	31	512
mfcr	Move From Condition Register	X	31	19
mfmsr	Move From Machine State Register	X	31	83
mfspr	Move From Special Purpose Register	X	31	339
mtspr	Move To Special Purpose Register	X	31	467
mul[o][.]	Multiply	XO	31	107
muls[o][.]	Multiply Short	XO	31	235
mtcrf	Move To Condition Register Fields	XFX	31	144
nabs[o][.]	Negative Absolute	XO	31	488
nand[.]	NAND	X	31	476
neg[o][.]	Negate	XO	31	104
nor[.]	NOR	X	31	124
or[.]	OR	X	31	444
orc[.]	OR With Complement	X	31	412
rrib[.]	Rotate Right And Insert Bit	X	31	537
sf[o][.]	Subtract From	XO	31	8
sfe[o][.]	Subtract From Extended	XO	31	36
sfme[o][.]	Subtract From Minus One Extended	XO	31	232
sfze[o][.]	Subtract From Zero Extended	XO	31	200
sl[.]	Shift Left	X	31	24
sle[.]	Shift Left Extended	X	31	153
sleq[.]	Shift Left Extended With MQ	X	31	217
sliq[.]	Shift Left Immediate With MQ	X	31	184
slliq[.]	Shift Left Long Immediate With MQ	X	31	248
sllq[.]	Shift Left Long With MQ	X	31	216
slq[.]	Shift Left With MQ	X	31	152
sr[.]	Shift Right	X	31	536
sra[.]	Shift Right Algebraic	X	31	792
srai[.]	Shift Right Algebraic Immediate	X	31	824
sraiq[.]	Shift Right Algebraic Immediate With MQ	X	31	952
sraq[.]	Shift Right Algebraic With MQ	X	31	920

Mnemonic	Instruction	Format	Primary Opcode	Extended Opcode
sre[.]	Shift Right Extended	X	31	665
srea[.]	Shift Right Extended Algebraic	X	31	921
sreq[.]	Shift Right Extended With MQ	X	31	729
sriq[.]	Shift Right Immediate With MQ	X	31	696
srliq[.]	Shift Right Long Immediate With MQ	X	31	760
srlq[.]	Shift Right Long With MQ	X	31	728
srq[.]	Shift Right With MQ	X	31	664
stbrx	Store Byte Reverse Indexed	X	31	662
stbux	Store Byte With Update Indexed	X	31	247
stbx	Store Byte Indexed	X	31	215
stfdux	Store Floating–Point Double With Update Indexed	X	31	759
stfdx	Store Floating–Point Double Indexed	X	31	727
stfsux	Store Floating–Point Single With Update Indexed	X	31	695
stfsx	Store Floating–Point Single Indexed	X	31	663
sthbrx	Store Half Byte Reverse Indexed	X	31	918
sthux	Store Half With Update Indexed	X	31	439
sthx	Store Half Indexed	X	31	407
stsi	Store String Immediate	X	31	725
stsx	Store String Indexed	X	31	661
stux	Store With Update Indexed	X	31	183
stx	Store Indexed	X	31	151
t	Trap	X	31	4
xor[.]	XOR	X	31	316
l	Load	D	32	
lu	Load With Update	D	33	
lbz	Load Byte And Zero	D	34	
lbzu	Load Byte And Zero With Update	D	35	
st	Store	D	36	
stu	Store With Update	D	37	
stb	Store Byte	D	38	
stbu	Store Byte With Update	D	39	
lhz	Load Half And Zero	D	40	

Mnemonic	Instruction	Format	Primary Opcode	Extended Opcode
lhzu	Load Half And Zero With Update	D	41	
lha	Load Half Algebraic	D	42	
lhau	Load Half Algebraic With Update	D	43	
sth	Store Half	D	44	
sthu	Store Half With Update	D	45	
lm	Load Multiple	D	46	
stm	Store Multiple	D	47	
lfs	Load Floating–Point Single	D	48	
lfsu	Load Floating–Point Single With Update	D	49	
lfd	Load Floating–Point Double	D	50	
lfdu	Load Floating–Point Double With Update	D	51	
stfs	Store Floating–Point Single	D	52	
stfsu	Store Floating–Point Single With Update	D	53	
stfd	Store Floating–Point Double	D	54	
stfdu	Store Floating–Point Double With Update	D	55	
fa[.]	Floating Add	A	63	21
fabs[.]	Floating Absolute Value	X	63	264
fcmpo	Floating Compare Ordered	X	63	32
fcmpu	Floating Compare Unordered	X	63	0
fd[.]	Floating Divide	A	63	8
fm[.]	Floating Multiply	A	63	5
fma[.]	Floating Multiply Add	A	63	29
fmr[.]	Floating Move Register	X	63	72
fms[.]	Floating Multiply Subtract	A	63	28
fnabs[.]	Floating Negative Absolute Value	X	63	136
fneg[.]	Floating Negate	X	63	40
fnma[.]	Floating Negative Multiply Add	A	63	31
fnms[.]	Floating Negative Multiply Subtract	A	63	30
frsp[.]	Floating Round To Single Precision	X	63	12
fs[.]	Floating Subtract	A	63	20
mcrfs	Move To Condition Register From FPSCR	X	63	64
mffs[.]	Move From FPSCR	X	63	583
mtfsb0[.]	Move To FPSCR Bit 0	X	63	70
mtfsb1[.]	Move To FPSCR Bit 1	X	63	38
mtfsf[.]	Move To FPSCR Fields	XFL	63	711
mtfsfi[.]	Move To FPSCR Field Immediate	X	63	134

Instruction Set, Indexed by Mnemonic

Mnemonic	Instruction	Format	Primary Opcode	Extended Opcode
a[o][.]	Add	XO	31	10
abs[o][.]	Absolute	XO	31	360
ae[o][.]	Add Extended	XO	31	138
ai	Add Immediate	D	12	
ai.	Add Immediate And Record	D	13	
ame[o][.]	Add To Minus One Extended	XO	31	234
and[.]	AND	X	31	28
andc[.]	AND With Complement	X	31	60
andil.	AND Immediate Lower	D	28	
andiu.	AND Immediate Upper	D	29	
aze[o][.]	Add To Zero Extended	XO	31	202
b[l][a]	Branch	I	18	
bc[l][a]	Branch Conditional	B	16	
bcc[l]	Branch Conditional To Count Register	XL	19	528
bcr[l]	Branch Conditional Register	XL	19	16
cal	Compute Address Lower	D	14	
cau	Compute Address Upper	D	15	
cax[o][.]	Compute Address	XO	31	266
cmp	Compare	X	31	0
cmpi	Compare Immediate	D	11	
cmpl	Compare Logical	X	31	32
cmpli	Compare Logical Immediate	D	10	
cntlz[.]	Count Leading Zeroes	X	31	26
crand	Condition Register AND	XL	19	257
crandc	Condition Register AND With Complement	XL	19	129
creqv	Condition Register Equivalent	XL	19	289
crnand	Condition Register NAND	XL	19	225
crnor	Condition Register NOR	XL	19	33
cror	Condition Register OR	XL	19	449
crorc	Condition Register OR With Complement	XL	19	417
crxor	Condition Register XOR	XL	19	193
div[o][.]	Divide	XO	31	331

Mnemonic	Instruction	Format	Primary Opcode	Extended Opcode
divs[o][.]	Divide Short	XO	31	363
doz[o][.]	Difference Or Zero	XO	31	264
dozi	Difference Or Zero Immediate	D	09	
eqv[.]	Equivalent	X	31	284
exts[.]	Extend Sign	X	31	922
fa[.]	Floating Add	A	63	21
fabs[.]	Floating Absolute Value	X	63	264
fcmpo	Floating Compare Ordered	X	63	32
fcmpu	Floating Compare Unordered	X	63	0
fd[.]	Floating Divide	A	63	8
fm[.]	Floating Multiply	A	63	5
fma[.]	Floating Multiply Add	A	63	29
fmr[.]	Floating Move Register	X	63	72
fms[.]	Floating Multiply Subtract	A	63	28
fnabs[.]	Floating Negative Absolute Value	X	63	136
fneg[.]	Floating Negate	X	63	40
fnma[.]	Floating Negative Multiply Add	A	63	31
fnms[.]	Floating Negative Multiply Subtract	A	63	30
frsp[.]	Floating Round To Single Precision	X	63	12
fs[.]	Floating Subtract	A	63	20
l	Load	D	32	
lbrx	Load Byte Reverse Indexed	X	31	534
lbz	Load Byte And Zero	D	34	
lbzu	Load Byte And Zero With Update	D	35	
lbzux	Load Byte And Zero With Update Indexed	X	31	119
lbzx	Load Byte And Zero Indexed	X	31	87
lfd	Load Floating–Point Double	D	50	
lfdu	Load Floating–Point Double With Update	D	51	
lfdux	Load Floating–Point Double With Update Indexed	X	31	631
lfdx	Load Floating–Point Double Indexed	X	31	599
lfs	Load Floating–Point Single	D	48	
lfsu	Load Floating–Point Single With Update	D	49	

Mnemonic	Instruction	Format	Primary Opcode	Extended Opcode
lfsux	Load Floating–Point Single With Update Indexed	X	31	567
lfsx	Load Floating–Point Single Indexed	X	31	535
lha	Load Half Algebraic	D	42	
lhau	Load Half Algebraic With Update	D	43	
lhaux	Load Half Algebraic With Update Indexed	X	31	375
lhax	Load Half Algebraic Indexed	X	31	343
lhbrx	Load Half Byte Reverse Indexed	X	31	790
lhz	Load Half And Zero	D	40	
lhzu	Load Half And Zero With Update	D	41	
lhzux	Load Half And Zero With Update Indexed	X	31	311
lhzx	Load Half And Zero Indexed	X	31	279
lm	Load Multiple	D	46	
lscbx[.]	Load String And Compare Byte Indexed	X	31	277
lsi	Load String Immediate	X	31	597
lsx	Load String Indexed	X	31	533
lu	Load With Update	D	33	
lux	Load With Update Indexed	X	31	55
lx	Load Indexed	X	31	23
maskg[.]	Mask Generate	X	31	29
maskir[.]	Mask Insert From Register	X	31	541
mcrf	Move Condition Register Field	XL	19	0
mcrfs	Move To Condition Register From FPSCR	X	63	64
mcrxr	Move To Condition Register From XER	X	31	512
mfcr	Move From Condition Register	X	31	19
mffs[.]	Move From FPSCR	X	63	583
mfmsr	Move From Machine State Register	X	31	83
mfspr	Move From Special Purpose Register	X	31	339
mtcrf	Move To Condition Register Fields	XFX	31	144
mtfsb0[.]	Move To FPSCR Bit 0	X	63	70
mtfsb1[.]	Move To FPSCR Bit 1	X	63	38
mtfsf[.]	Move To FPSCR Fields	XFL	63	711
mtfsfi[.]	Move To FPSCR Field Immediate	X	63	134

Mnemonic	Instruction	Format	Primary Opcode	Extended Opcode
mtspr	Move To Special Purpose Register	X	31	467
mul[o][.]	Multiply	XO	31	107
muli	Multiply Immediate	D	07	
muls[o][.]	Multiply Short	XO	31	235
nabs[o][.]	Negative Absolute	XO	31	488
nand[.]	NAND	X	31	476
neg[o][.]	Negate	XO	31	104
nor[.]	NOR	X	31	124
or[.]	OR	X	31	444
orc[.]	OR With Complement	X	31	412
oril	OR Immediate Lower	D	24	
oriu	OR Immediate Upper	D	25	
rlimi[.]	Rotate Left Immediate Then Mask Insert	M	20	
rlinm[.]	Rotate Left Immediate Then AND With Mask	M	21	
rlmi[.]	Rotate Left Then Mask Insert	M	22	
rlnm[.]	Rotate Left Then AND With Mask	M	23	
rrib[.]	Rotate Right And Insert Bit	X	31	537
sf[o][.]	Subtract From	XO	31	8
sfe[o][.]	Subtract From Extended	XO	31	36
sfi	Subtract From Immediate	D	08	
sfme[o][.]	Subtract From Minus One Extended	XO	31	232
sfze[o][.]	Subtract From Zero Extended	XO	31	200
sl[.]	Shift Left	X	31	24
sle[.]	Shift Left Extended	X	31	153
sleq[.]	Shift Left Extended With MQ	X	31	217
sliq[.]	Shift Left Immediate With MQ	X	31	184
slliq[.]	Shift Left Long Immediate With MQ	X	31	248
sllq[.]	Shift Left Long With MQ	X	31	216
slq[.]	Shift Left With MQ	X	31	152
sr[.]	Shift Right	X	31	536
sra[.]	Shift Right Algebraic	X	31	792
srai[.]	Shift Right Algebraic Immediate	X	31	824

Mnemonic	Instruction	Format	Primary Opcode	Extended Opcode
sraiq[.]	Shift Right Algebraic Immediate With MQ	X	31	952
sraq[.]	Shift Right Algebraic With MQ	X	31	920
sre[.]	Shift Right Extended	X	31	665
srea[.]	Shift Right Extended Algebraic	X	31	921
sreq[.]	Shift Right Extended With MQ	X	31	729
sriq[.]	Shift Right Immediate With MQ	X	31	696
srliq[.]	Shift Right Long Immediate With MQ	X	31	760
srlq[.]	Shift Right Long With MQ	X	31	728
srq[.]	Shift Right With MQ	X	31	664
st	Store	D	36	
stb	Store Byte	D	38	
stbrx	Store Byte Reverse Indexed	X	31	662
stbu	Store Byte With Update	D	39	
stbux	Store Byte With Update Indexed	X	31	247
stbx	Store Byte Indexed	X	31	215
stfd	Store Floating–Point Double	D	54	
stfdu	Store Floating–Point Double With Update	D	55	
stfdux	Store Floating–Point Double With Update Indexed	X	31	759
stfdx	Store Floating–Point Double Indexed	X	31	727
stfs	Store Floating–Point Single	D	52	
stfsu	Store Floating–Point Single With Update	D	53	
stfsux	Store Floating–Point Single With Update Indexed	X	31	695
stfsx	Store Floating–Point Single Indexed	X	31	663
sth	Store Half	D	44	
sthbrx	Store Half Byte Reverse Indexed	X	31	918
sthu	Store Half With Update	D	45	
sthux	Store Half With Update Indexed	X	31	439
sthx	Store Half Indexed	X	31	407
stm	Store Multiple	D	47	
stsi	Store String Immediate	X	31	725
stsx	Store String Indexed	X	31	661
stu	Store With Update	D	37	

Mnemonic	Instruction	Format	Primary Opcode	Extended Opcode
stux	Store With Update Indexed	X	31	183
stx	Store Indexed	X	31	151
svc[l][a]	Supervisor Call	SC	17	
t	Trap	X	31	4
ti	Trap Immediate	D	03	
xor[.]	XOR	X	31	316
xoril	XOR Immediate Lower	D	26	
xoriu	XOR Immediate Upper	D	27	

Index

Symbols

A

B

C

D

H

I

L

M

N

O

P

R

S

T

V

X

Reader's Comment Form

Assemble Langauge Reference

SC23–2197–00

Please use this form only to identify publication errors or to request changes in publications. Your comments assist us in improving our publications. Direct any requests for additional publications, technical questions about IBM systems, changes in IBM programming support, and so on, to your IBM representative or to your IBM-approved remarketer. You may use this form to communicate your comments about this publication, its organization, or subject matter, with the understanding that IBM may use or distribute whatever information you supply in any way it believes appropriate without incurring any obligation to you.

☐ If your comment does not need a reply (for example, pointing out a typing error), check this box and do not include your name and address below. If your comment is applicable, we will include it in the next revision of the manual.

☐ If you would like a reply, check this box. Be sure to print your name and address below.

Page	Comments

Please contact your IBM representative or your IBM–approved remarketer to request additional publications.

Please print

Date _____

Your Name _____

Company Name _____

Mailing Address _____

_____

_____

Phone No. ( ) _____
Area Code

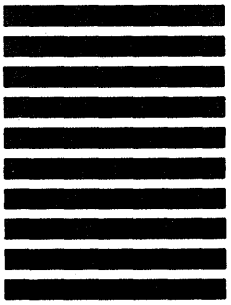
No postage necessary if mailed in the U.S.A

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 997
11400 Burnet Rd.
Austin, Texas 78758–3493

Fold

Fold

Cut or Fold Along Line

Fold and Tape

Please Do Not Staple

Fold and Tape

International Business Machines
Corporation
11400 Burnet Road
Austin, Texas 78758–3493

Printed in the
United States of America

SC23-2197-00