IBM TECHNICAL NEWSLETTER

for the

RT Personal Computer

Assembler Language Reference

—OVER—

Order Numbers:
74X1025
SN20-9798
26 September 1986

TB74X1025
Printed in U.S.A.

Summary of Changes

This Technical Newsletter contains new information about shared libraries.

Perform the following:

Remove Pages	**Insert Update Pages**
ix, x	ix, x
4-129, 4-130	4-129, 4-130
5-13 through 5-18	5-13 through 5-18

Note: Please file this cover letter at the back of the manual to provide a record of changes.

26 September 1986

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

Ordering Additional Copies of This Book

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

- To order from your IBM representative, use Order Number SV21-8011.
- To order from your IBM dealer, use Part Number 75X1024.

A binder is included with the order.

Shift Algebraic Right Immediate **sari**

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

Format: sari R1,I2

small form

A0	R1	IN
0	8	12 15

large form

A1	R1	IN
0	8	12 15

Remarks:

- I2 must evaluate to an integer between decimal 0 and 31 inclusive.
- The assembler examines I2 and generates the correct form (small or large) of the instruction. If I2 ≤ 15, then IN = I2, and op code A0 is generated. If I2 > 15, then IN = I2 - 16, and op code A1 is generated.
- Condition Status bits LT, EQ, and GT are affected.

Examples:

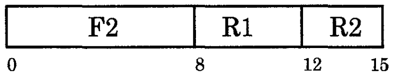
```
   # assume GPR 4 holds 0x1234 5678
sari 4,8
   # op code A0 is generated
   # now GPR 4 holds 0x0012 3456
   # GT bit set to one


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

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

Purpose: The one's complement of the content of register R2 is added to the content of register R1. The value of Condition Status bit C0 is added to the result. The final result is placed in register R1.

Format: se R1,R2



Remarks:

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

Example:

```
   # assume GPR 4 holds 0x0044 6655
   # assume GPR 5 holds 0x0033 4422
setcb 0xC   # C0 now set to one
se 4,5
   # now GPR 4 holds 0x0011 2233
   # GT and C0 bits set to one
```

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

Purpose: Assemble the values represented by the **exp** expressions into consecutive bytes.

Format: .byte exp,exp, ...

Remarks:

- The **exps** cannot contain externally defined symbols.
- If an **exp** is longer than one byte, it will be truncated.

Example:

```
                  .set olddata,0xCC
                       .
                       .
@2000 0000  mine: .byte 0x3F,0x7+0xA,olddata,0xFF
                       # load GPR 1 with 0x20000000
                  l 2,0(1)
                       # GPR 2 now holds 0x3F11CCFF
```

call

Purpose: Calls a subroutine. The label is the name of the subroutine being called. If the label is a C language subroutine, the label includes the leading period (.).

The pcp_address is the hex address of the pointer to the called routine's constant pool. This operand can be expressed as a label, or as a base and displacement of the form **D2(R2)**. However, any labels specified must be covered by a **.using**.

The number_words is the number of words required to store all parameters passed between the calling and the called routine. This value is used only by debuggers such as **sdb**. The debugger uses this value to display procedure parameters when showing information about the call. If debugger information is not being collected, this value is zero.

Format: call label, pcp_address, number_words

Remarks:

- The assembler expands the **call** pseudo-op into the following series of statements:

  ```
  balix  15, .label          # call the routine
  l      0, pcp_address      # get the routine's constant pool pointer
  .byte  0x08, number_words  # number of words of parameters passed;
                             # .byte acts as a no-op with operands
  ```

  The **.byte** statement is generated only when the **number_words** operand was not zero (that is, only when **sdb** information was being gathered). If **number_words** was zero, then the **.byte** statement is not generated.

  If the target of the **balix** is not within a megabyte of the call, then the branch target cannot be resolved. This could happen if the load module's text segment is larger than a megabyte, or if the target is in a shared library that is mapped into some other segment. In either case, the linkage editor changes **balix** into **balax** to a special sequence of code at a fixed location in the RT PC kernel.

  This special linkage code uses the first word of the called routine's constant pool to derive the address of the distant entry point. If **ld** has set the low-order bit of the first constant pool word to zero, then the linkage code assumes that the word is the address of the entry point. So, the linkage code branches to that address.

  If **ld** has set the low-order bit of the first constant pool word to one, then the linkage code assumes that the word is the address of an entry in the "gate vector" at the beginning of a shared library text image. Each entry in this gate vector represents the offset from the beginning of the text image of a function contained in the library. The linkage code adds this entry (the function offset) to the shared library's starting location, then branches to the resulting address.

The special linkage routine in the kernel looks like this:

```
KLRTN:              # KLRTN's absolute address is 0x0c00
 mts   10,14        # save register 14 into MQ register
 lr    14,0         # make register 0, the pcp, addressable
 l     14,0(14)     # get entry address
 mttbi 14,31        # check low-order bit
 bts   fixup        # if low-order bit is one, go to fixup pointer
 brx   14           # if low-order bit zero, go to real entry point
 mfs   10,14        # restore register 14 to its previous value


fixup:     # "fixup" the address you were branching to --
           # assume register 14 holds a pointer to a long
           # word containing the offset within a shared library
           # text image of the real target entry point;
           # register 14 also contains the segment number
           # where the shared image resides
           # in the high 4 bits

 nilo 14,14,0xfffe   # set lowest bit of register 14 to zero
 st   15,-4(1)       # save register 15 -- eventually, called
                     # routine will return to register 15
 niuz 15,14,0xf000   # get segment number (high-order 4 bits of
                     # the address ld used)
 l    14,0(14)       # de-reference register 14 (holds pointer
                     # into gate vector)
 o    14,15          # "or" the segment number (high-order 4 bits)
                     # into the address held by register 14
 lr   15,0           # get pcp again
 st   14,0(15)       # register 14 now has real address of program,
                     # so save that address -- next time this
                     # program is called, low-order bit of first
                     # constant pool word will be zero
 l    15,-4(1)       # restore register 15
 brx  14             # continue the call
 mfs  10,14          # restore register 14
```

- The linkage sequence assumes that register 0 points to the constant pool of the called routine, and that the first word of the constant pool is the address of the routine's entry point.

Example: The assembler expands this:

```
call  .foo, 12(14),3
```

into this:

```
balix  15,.foo
l      0,12(14)
.byte  0x08,3
```

See Also: "Subroutine Linkage and System Calls" on page 6-10

AIX Operating System Programming Tools and Interfaces and **shlib** in *AIX Operating System Commands Reference* for information about gate vectors

callr

Purpose: Calls a subroutine using a register that holds a pointer to the called routine's constant pool.

R is the register containing the address of the called routine's constant pool. R must not be 0. Number_words is the number of words required to store all parameters passed between the caller and the called routine. This value is used only by debuggers such as **sdb**. The debugger uses this value to display procedure parameters when showing information about the call. If debugger information is not being collected, this value is zero.

Format: callr R, number_words

Remarks:

- It is impossible to predict what function address will be loaded into register R at execution time. Therefore, the assembler must generate code to suit the worst case: a call to a function within a shared library. Accordingly, the assembler expands **callr** into a series of statements including a branch to KLRTN, the special kernel linkage routine mentioned on page 5-15:

  ```
  balax KLRTN        # branch to kernel linkage routine --
                     # KLRTN is its absolute address in memory
  lr    0,R          # put address of called routine's
                     # constant pool into register 0
  .byte 0x08, number_words # number of words of parameters passed;
                           # .byte acts as a no-op with operands
  ```

  The **.byte** statement is generated only when the **number_words** operand was not zero (that is, only when **sdb** information was being gathered). If **number_words** was zero, then the **.byte** statement is not generated.

- The linkage sequence assumes that register 0 points to the constant pool of the called routine, and that the first word of the constant pool is the address of the routine's entry point.

Example: The assembler expands this:

```
callr  8,3
```

into this:

```
balax KLRTN  # KLRTN is the special kernel linkage routine
             # described on page 5-15
lr    0,8    # get the pointer to called routine's constant pool
.byte  0x08,3   # three words of parameters are passed
```

See Also: "Subroutine Linkage and System Calls" on page 6-10

.comm

Purpose: Define a block of storage that will be common to more than one module. The block is named **name** and has a length of **exp** bytes.

Format: .comm name,exp

Remarks:

- The **exp** operand must be an absolute expression; **name** is relocatable.
- Use **.comm** when you know the size of a block of data that will be shared by two or more files, but you don't know whether that data will become initialized.
- The linker defines a common block of storage at link time. That is, the space declared with a **.comm** disappears at link time. If the data in the **.comm** space becomes initialized, it goes to the data runtime segment. If the **.comm** data is not initialized, it goes to the bss section. At load time, the bss section is created at the end of the data segment.
- If the original module or any linked modules contain more than one **.comm** definition of the same **name**, the assembler reserves space specified by the largest **exp**. The assembler does not generate an error message.
- By default, the linker defines common blocks in the bss section of the linked program. If you link in a module that defines **name** in the text or data assembler section, that module's definition of **name** will take precedence. The common block will then be defined in the text or data assembler section.

Example:

```
.comm  proc,5120
    # if proc is not defined elsewhere, proc
    # refers to 5120 bytes of storage in
    # the bss segment of the linked program
```

See Also: .data, .globl, .lcomm, .text

Chapter 3

Notes

Notes

Notes

Notes