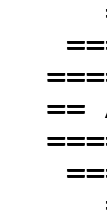
# RAG SOFTWARE

# AEMS MACRO ASSEMBLER

# MACRO DESCRIPTIONS AND MACRO WRITING




Macro Assembler
Version 1
R. A. Green

**CONTENTS**





January 1993

## MACRO DESCRIPTIONS

The following sections describe the use of the macros in the ARAGMAC macro library. These are general purpose macros.

```
[label] BE     gad                    Branch Equal
[label] BNE    gad                    Branch Not Equal
[label] CALL   gad,p1,p2,p3,p4        Call Subroutine
[label] IF     gas,relop,gad,target   IF word
[label] IFB    gas,relop,gad,target   IF Byte
[label] IFSW   gas,{ON|OFF},target    IF Switch
[label] LDB    gas,wad                Load Byte
[label] MOVBL  gas,gad,length         MOVe Bytes Long
[label] RCALL  gad,R0=p0,R1=p1,R2=p2  Call Subroutine
[label] SETSW  gad,{ON|OFF}           Set Switch
[label] SETV   gas[,wad][,NOP]        SET Vdp address
```

**BE - Branch Equal**

```
     [label] BE     destination      [comment]
```

Causes a branch to the destination if the EQ status is set. The destination operand is coded as a general address.
Examples:

```
             BE     @GOOD
     TEST    BE     *R8
```

**BNE - Branch Not Equal**

```
     [label] BNE    destination      [comment]
```

Causes a branch to the destination if the EQ status is not set. The destination operand is coded as a general address.
Examples:

```
             BNE    @GOOD
     TEST    BNE    *R8
```

**CALL - Call Subroutine**

```
     [label] CALL   destination,p1,p2,p3,p4  [comment]
```

Causes a BLWP to the destination subroutine and generates a parameter list following the BLWP for the parameters "p1" to "p4". All parameters are optional and DATA statements are generated only for as many parameters as are specified.
Examples:

```
             CALL   @DSRLNK,8
     CSUB1   CALL   @SUB1,PARM1,PARM2,PARM3
```

**IF - IF Word Jump**

```
     [label] IF     source,relop,destination,target  [comment]
```

Compares the word at "source" to the word at "destination" then jumps to label "target" depending upon the "relop". The relational operator "relop" is any of the jump conditions (i.e. EQ for JEQ, H for JH, etc.). The source and destination operands are coded as general addresses, OR the destination may be coded as a literal value and the source as a register value. A literal is written as an equals sign followed by a self defining value. When a literal destination is coded, a CI instruction is generated.
Examples:

```
             IF     @X,EQ,*R5+,EQUAL
     TEST    IF     *R3,GT,@Y,GREAT
             IF     R4,EQ,=10,EQUAL     R4 = 10?
```

**IFB - IF Byte Jump**

```
     [label] IFB    source,relop,destination,target  [comment]
```

Compares the byte at "source" to the byte at "destination" then jumps to label "target" depending upon the "relop". The relational operator "relop" is any of the jump conditions (i.e. EQ for JEQ, H for JH, etc.). The source and destination operands are coded as general addresses.
Examples:

```
        IFB    @X,EQ,*R5+,EQUAL
TEST    IFB    *R3,GT,@Y,GREAT
```

**IFSW - IF Switch Jump**

```
[label] IFSW   source,{ON|OFF},target  [comment]
```

Tests the switch "source" and jumps to label "target" depending upon whether the switch is ON or OFF. The source operand is coded as the general address of the switch word. A switch is ON if the value of the word is non-zero and is OFF if the word is zero. Switches can be set with the SETSW macro described later.
Examples:

```
        IFSW   @SW1,ON,SW1ON
TEST    IFSW   *R3,OFF,SKIP
```

**LDB - Load Byte**

```
[label] LDB    source,destination
```

Loads the "source" byte as a word into the "destination" register. The "source" operand is specified as a general address. The "destination" operand must be a workspace register.
Examples:

```
LOAD    LDB    @A,R1
        LDB    R1+,R4
```

**MOVBL - Move Bytes Long**

```
[label] MOVBL  source,destination,length  [comment]
```

Moves the bytes from "source" to "destination". The number of bytes moved is specified by "length" which may be the general address of the word containing the length or may be a literal length. A literal is written as an equals sign followed by a self defining value. This macro generates a loop to move the bytes. The "source" and "destination" operands are coded as general addresses. If "source" is specified as a symbolic memory reference then R0 is used. If "destination" is specified as a symbolic memory reference then R1 is used. If "length" is specified as a symbolic memory reference or a literal then R2 is used.
Examples:

```
        MOVBL  @X,@Y,@LEN
MXY     MOVBL  @X,@Y,=32
        MOVBL  *R5+,*R7+,R3
        MOVBL  *R10,@X,=20
```

## RCALL - Call Subroutine

```
[label] RCALL  destination,R0=p0,R1=p1,R2=p2  [comment]
```

Causes the specified registers to be loaded using LI instructions and a BLWP to the "destination" subroutine. All parameters are optional and LI instructions are generated only for those parameters specified.
Examples:

```
        RCALL  @VMBW,R0=VADDR,R1=CADDR,R2=20
VREAD   RCALL  @VSBR,R0=>0020
```

## SETSW - Set Switch

```
[label] SETSW  gad,{ON|OFF}
```

Sets the switch word specified by the general address "gad" ON or OFF. The switch value for OFF is zero (set via the CLR instruction). The switch value for ON is non-zero (set by the SETO instruction).
Examples:

```
        SETSW  @SW1,ON
SET     SETSW  *R3,OFF
```

## SETV - Set VDP Address

```
[label] SETV   gas[,wad],[NOP]
```

Sets the VDP address specified by "gas". If gas is a symbolic memory address or a literal then R0 is used. If specified, "wad" is a register that has the VDP write address address,>8C02. If specified, NOP, indicates that a NOP instruction should be generated after the VDP address is set.
Examples:

```
        LI     R15,>8C02
        SETV   =>0C00,R15
SET     SETV   R3
        SETV   R3,R15,NOP
```

## DEVELOPING MACROS

The macro library is a DISPLAY VARIABLE 80 file and can be edited the same as an assembler source file. Note that the order of the macros in the library is not important. When developing a new macro you should place the macro definition in your source file. Then when you are satisfied that the macro works you should copy it to the macro file.

All macro definitions are kept in memory during assembly so that the size and number of macro statements is limited. When you copy your macro definitions to the macro library you should "compress" them. That is you should remove all unnecessary blanks and all comments. You should periodically review your use of macros and remove from the library any that you find you are not using.

The following discussion of macros is in the nature of a tutorial on what a macro is, and how to develop one.

Let us try some analogies in order to get a feeling for what macros are and how to make use of them. Like most analogies, none of them are completely accurate.

1. Macros are like subroutines, (generally very small subroutines),
2. Macros are like COPY statements,
3. Macros are like user defined functions in BASIC (i.e. DEF SQ(X)=X*X),
4. Macros are like small programs that generate assembler source statements.

First, we will look at analogy number 1. Suppose that in a program you had many places where you want to move 10 bytes of data from one place in memory to another. You could code a "MOVE" subroutine as shown below.

```
Line/Pos  12345678901234567890123456789012345678901234567890
001       *...
002               BL   @MOVE
003       *...
004               BL   @MOVE
005       *...
006       MOVE    LI   R0,X          move 10 from x to y
007               LI   R1,Y
008               LI   R2,10
009               MOVB *R0+,*R1+
010               DEC  R2
011               JGT  $-4
012               RT
013       X       BSS  10
014       Y       BSS  10
015               END
```

Now, if you wanted extreme speed you would repeat the code in lines 6 through 11 at line 2 then again at line 4 and so on for every reference to MOVE. If you used your move subroutine a lot of times in the program, this repetition would get tiresome. With the macro assembler, you could define a macro and use it wherever the move was needed. For example:

```
Line/Pos  12345678901234567890123456789012345678901234567890
001       $MACRO MOVE               BEGINNING OF DEFN
002              LI   R0,X          move 10 from x to y
003              LI   R1,Y
004              LI   R2,10
005              MOVB *R0+,*R1+
006              DEC  R2.
007              JGT  $-4
008       $END                      END OF DEFN
009       *...
010              MOVE
011       *...
012              MOVE
013       *...
014       X      BSS  10
015       Y      BSS  10
016              END
```

When this code is assembled, the assembler stores away the macro definition in lines 1 to 8, then later when it sees the macro name, MOVE, used as an operation code in lines 10 and 12, it substitutes the statements inside the macro definition into the source.

Using analogy number 2, the above could be accomplished by putting lines 2 through 7 into a file, say DSK1.MOVE and then recoding the program as shown below.

```
Line/Pos  12345678901234567890123456789012345678901234567890
009       *....
010              COPY "DSK1.MOVE"
011       *...
012              COPY "DSK1.MOVE"
013       *...
014       X      BSS  10
015       Y      BSS  10
016              END
```

At this point, you should enter the above two source programs and assemble them. Use options R (for registers), L (for listing) and G (for show generated statements). Compare the two listings and see what statements were actually assembled. Note on the listing that plus signs precede statements generated by a macro.

The above example of the use of a macro is very trivial. So trivial that you probably would not use a macro for this purpose. Let's extend the example a bit more.

Suppose, that each time you wanted to use MOVE, you wanted to move different strings of different lengths. Say the first time you wanted to move 20 bytes of A to B and the second time you wanted to move 10 bytes of X to Y. In this case, the "COPY" solution would not work since the code copied from the file is always the same, but it can be done with a macro.

Macros are like subroutines (i.e. CALL COLOR(...) in BASIC) in that they can have parameters. One parameter is the symbol you code in the label field of the macro statement. There are a maximum of nine other parameters which are the values coded in the operand field of the macro statement. The macro parameters have fixed "special" names. The symbol "&P0" is the name of the label field parameter, "&P1" is the name of the first operand, "&P2" the name of the second operand, and so on up to "&P9". When the assembler sees one of these special names within a macro definition it substitutes the value specified on the macro statement. Thus each time you code a macro statement (i.e. every time you use the macro name as an operation code) with different operands, different values are substituted and different statements are "generated" by the macro. Let's now recode our example.

```
Line/Pos  12345678901234567890123456789012345678901234567890
001       $MACRO MOVE                BEGINNING OF DEFN
002       &P0    LI   R0,&P1           &P1 IS FROM
003              LI   R1,&P2           &P2 IS TO
004              LI   R2,&P3           &P3 IS LENGTH
005              MOVB *R0+,*R1+
006              DEC  R2
007              JGT  $-4
008       $END                        END OF DEFN
009       *...
010              MOVE A,B,20
011       *...
012       NEXT   MOVE X,Y,10
013       *...
014       A      BSS  20
015       B      BSS  20
016       X      BSS  10
017       Y      BSS  10
018              END
```

At this point you should enter and assemble the above example using the RLG options. In the listing you can see that "A" was substituted for "&P1" in the first generation for MOVE and that "X" was substituted in the second generation. Note also that the label coded on line 12 is substituted for "&P0".

Now our macro has become less trivial and considerably more useful. It is still a simple use of macros. Let's take our macro one step Further. Suppose that the length of the move in line 10 was calculated by some other part of the program and was stored at "LNGT". Then we would want the macro to generate the statement

```
MOV  @LNGT,R2.
```

instead of

```
LI   R2,20
```

for line 10, but to generate the same code as before for line 12. The following example accomplishes this.

```
Line/Pos  12345678901234567890123456789012345678901234567890
001       $MACRO MOVE               BEGINNING OF DEFN
002       &P0    LI   R0,&P1          &P1 IS FROM
003              LI   R1,&P2          &P2 IS TO
004       $IF    '&P3(1.1)',EQ,'@',GENMOV
005              LI   R2,&P3          &P3 IS LENGTH
006       $GOTO  COMMON
007       $LABEL GENMOV
008              MOV  &P3,R2          &P3 IS @LENGTH
009       $LABEL COMMON
010              MOVB *R0+,*R1+
011              DEC  R2
012              JGT  $-4
013       $END                        END OF DEFN
014       *...
015              MOVE A,B,@LNGT
016       *...
017       NEXT   MOVE X,Y,10
018       *...
019       A      BSS  20
020       B      BSS  20
021       X      BSS  10
022       Y      BSS  10
023       LNGT   BSS  2
024              END
```

Again, you should enter and assemble this example.

This example shows how macros are like programs that generate assembler statements (analogy number 4). At line 4 in the macro definition, the $IF macro directive tests if the first character of the third operand is an at sign (indicating that the number of bytes to move is in storage). If it is an at sign, the MOV instruction at line 8 is generated, otherwise, the LI at line 5 is generated and the $GOTO directive causes generation to continue with the code common to both options.

Line 4 uses "substring notation". The bracketed numbers following &P3 tell the assembler to substitute only part of the third operand. Substring notation is similar to the BASIC

SEG$(&P3,1,1). Note, however, that a period is used between the two numbers.

We now have a fairly complex macro that could be useful in many different programs. You could at this point put the definition (lines 1 to 13) in your macro library and then use the MOVE macro in all your programs just as though it were an ordinary operation code.

Let's not stop yet though. Let's extend the macro some more. First, read the description of the MOVBL macro given earlier in the manual. Now, modify your MOVE macro to function like MOVBL. Give it some thought before you peek at the definition given in the macro library on the assembler disk. Try assembling a few MOVBL macros (with the RGL options) and see what code is generated.