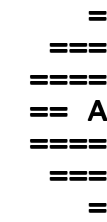
RAG SOFTWARE

AEMS MACRO ASSEMBLER

ASSEMBLER LANGUAGE REFERENCE



Macro Assembler
Version 1.1
R. A. Green

CONTENTS

April 1993

COMPATIBILITY

The AEMS Macro Assembler, with the exceptions noted below, is compatible with the Assembler Language described in the TI Editor/Assembler manual for the TI 99/4A. It is also upward compatible with the RAG Software TI 99/4A Macro Assembler.

Unsupported Features

The following Instructions are not assembled: CKON, CKOF, IDLE, LREX, RSET.

The following Assembler Directives are not supported: CEND, CSEG, DEND, DSEG, DXOP, LOAD, PEND, PSEG, SREF.

The DORG directive is supported only with an absolute expression as the operand.

Extensions

The AEMS Macro Assembler has several extensions to the TI Assembler Language and syntax.

First, the macro facility is a major extension. The macro facility includes the ability to define new instruction operation codes as well as define macro instructions.

Second, six new assembler directives: EQUV, OBJREC, FLOAT, COMMON, PNUM and STRI have been added to the language.

Third, a quoted string may use either single or double quotes as the string delimiter. In either case, if the delimiter is to occur within the string it must be represented by a pair. A string may also specified as a sequence of hexadecimal digits preceded by the ">" usually used to indicate hex constants.

Fourth, COPY statements may use an asterisk as the device name or as the disk number. This is used to tell the assembler that the copy file is on the same device/directory or the same disk number as the source file. For example:

```
COPY 'DSK*.PARTB'
COPY "*.PARTC"
```

Both are valid copy statements indicating that the file PARTB is on the same disk number as the source file and that PARTC is on the same device/directory as the source file.

Fifth, the characters, number sign (#), dollar sign ($), percent sign (%) and underscore (_) may be used as the second or following characters of assembler symbols.

Sixth, complex relocatable address values are allowed. A complex relocatable value is a reference to a COMMON defined symbol or to a REF symbol plus an offset.

ELEMENTS OF THE LANGUAGE

In order to understand and use the assembler language there are a number of definitions and conventions that must be understood.

Assembler Statements

Each line of Assembler code is called a statement. There are five types of statements, each of which is defined below.

COMMENT -- these statements provide notes for the person reading the code. The assembler ignores comments except for printing them in the listing. Comment statements are identified by an asterisk in position one of the statement.

ASSEMBLER DIRECTIVES -- these statements give the assembler directions on how you want your code assembled. Assembler directive statements have the following format.

```
[label] operation operands [comment].
```

Each of the four fields in the statement are separated by one or more blanks or spaces. The label and comment fields are always optional. The label if present must begin in position one of the statement. If no label is coded, at least one blank must precede the operation field. Some assembler directives have no operands in which case the comment field immediately follows the operation field. Individual operands within the operands field are separated from each other by commas. No blanks must occur within the operand field unless the operand is enclosed in quotes. The operation field names the assembler directive. All the assembler directives are described later.

MACRO DIRECTIVES -- these statements give the assembler directions on how to interpret and assemble your macros. Macro directives occur only within a "macro definition". NOTE that the $OPCODE directive is described as a macro directive although strictly it is not. Macro directives have the format shown below.

```
$operation operands [comment]
```

Each of the three fields in the statement are separated by one or more blanks or spaces. Macro directives are recognized by the dollar sign coded in position one of the statement. The comment field is always optional. Some macro directives have no operands, in which case the comment field immediately follows the operation field. Individual operands within the operands field are separated from each other by commas. No blanks must occur within the operand field unless the operand is enclosed in quotes. The operation field names the macro directive. All the macro directives are described later.

ORDINARY STATEMENTS -- these statements represent machine instructions which are to be assembled. The bulk of your code will

be ordinary statements. Ordinary statements have the following format.

```
[label] operation operands [comment]
```

Each of the four fields in the statement are separated by one or more blanks or spaces. The label and comment fields are always optional. The label if present must begin in position one of the statement. If no label is coded, at least one blank must precede the operation field. Individual operands within the operands field are separated from each other by commas. No blanks must occur within the operand field unless the operand is enclosed in quotes. The operation field names the machine instruction to be assembled. All of the predefined machine instructions are described later. Some machine instructions have no operands in which case the comment field immediately follows the operation field. The $OPCODE directive can be used to define new machine instructions.

MACRO STATEMENTS -- these statements cause a macro to be invoked. Statements "generated" by a macro are assembled as though they appeared in the source file. Macro statements look just like ordinary statements as shown below.

```
[label] operation operands [comment]
```

The operation field names the macro definition that is to be used. The interpretation of the label field and the operands field is completely controlled by the macro definition.

Assembler Symbols

There are two kinds of assembler symbols.

ORDINARY SYMBOLS -- these symbols represent memory addresses or data values. Ordinary symbols are defined by appearing in the label field of an ordinary statement, an assembler directive statement or in the operand field of a REF directive. The value of an ordinary symbol is a 16-bit unsigned number. Unless otherwise specified, the value assigned to a symbol is the current location counter at which the statement is assembled.

Ordinary symbols are 1 to 6 characters in length. The first character must be a letter, "A" to "Z". The second and following characters can be a letter (A-Z), a number (0-9) or one of the characters "$", "#", "%" or "_".

MACRO SYMBOLS -- these are special predefined symbols used within a macro definition. The value of a macro symbol is a character string with a length of 0 to 60 characters. The names of the macro symbols are of the form &tn. Where the ampersand identifies a macro symbol. If you wish to code an ampersand that is not part of a macro symbol name within a macro definition you must code a pair of ampersands. The "t" in the macro symbol name is the type of symbol. There are four types of macro symbols: "P" for Parameter macro symbol, "L" for Local macro symbol, "G" for Global

macro symbol and "S" for System macro symbol. The "n" in the macro symbol name is a single digit from 0 to 9. Thus each type has ten different symbols and there are 40 macro symbols in total.

Parameter macro symbols have as their values the label field and the operands of the macro statement that invoked the macro. &P0 contains the label field, &P1 contains the first operand, &P2 contains the second operand, and so on. A macro statement can therefore have a maximum of 9 operands.

System macro symbols have values assigned by the assembler. These are:

- &S0 = value from the "Options" field.
- &S1 = the number of macros processed so far in the assembly. This value is useful for generating unique names within macros. The number is represented as a five character string with leading zeros.
- &S2 = the number of operands on the macro statement. The number is represented as a five character string with leading zeros.
- &S3 = a single character, "1" indicating the first pass of the assembler and "2" indicating the second pass of the assembler.
- &S4 = the information entered in the "Date" field.
- &S5 = the source file name.

The remainder of the system macro symbols are currently not used and have a null value.

Local macro symbols have values set via the $SET macro directive. All local macro symbols are reset to null at the beginning of each macro invocation.

Global macro symbols have values set via the $SET macro directive. All global macro symbols are reset to null at the beginning of each pass of the assembler. Global symbols can be used to communicate from one macro invocation to another within the same assembler pass.

Macro Symbol Substring Notation

Substrings of the macro symbol values are allowed. The general form for a macro symbol with substring notation is: &tn(s.l). Where "s" is the starting position for the substring and "l" is the length of the substring. Note that "s" and "l" are separated by a period not a comma.

As is usual for substring notation, if "s" specifies a position past the end of the string a null string will result. Also, if "l" specifies a length greater than the remainder of the string only the remainder is used. The "l" and the period are optional. If only "s" is specified then the remainder of the string is used.

Assume that the macro symbol &L2 has the value 'ABCDEFGHIJKLMNOPQRSTUVWXYZ', then:

```
&L2(25)    has the value 'YZ'
&L2(1.4)   has the value 'ABCD'
&L2(24.8)  has the value 'XYZ'
&L2(27.8)  has the value ''
```

There are cases where you may want a macro symbol to be followed by a bracketed expression that is not substring notation (i.e. in an indexed symbolic memory reference). This can be done by following the macro symbol with a period such as: &L2.(2). In fact any period following a macro symbol will be considered part of the macro symbol name and will be removed when the value of the macro symbol is substituted.

Macro Definitions

The macro facility gives you a shorthand way of coding assembler language programs. It can also be thought of as providing you with a slightly higher level of language (i.e. a language level somewhere between pure assembler and, say, BASIC). Usually, coding a single macro statement will cause several ordinary assembler statements to be generated and assembled into your program.

If you find yourself repeatedly coding the same group or sequence of statements with only slight differences, these could be coded within a macro definition and then replaced in your source programs by a single macro statement. This reduction in the number of statements in your program has several advantages. The source program is smaller thus easier to read and understand. Once the code generated by the macro is debugged then you need not debug each occurrence of it in your program. Less statements means less typing and less errors.

Macro definitions can be placed in the macro file or can be placed in the source file. Macro definitions in the source file must precede the first use of the macro.

A macro definition consists of macro directives, ordinary assembler statements or assembler directives. No macro statements may occur within a macro definition. During macro processing, the macro directives are executed by the assembler. Ordinary assembler statements and assembler directives are scanned and any macro symbols are replaced by their values. After replacement of macro symbols, the ordinary statements and assembler directives are assembled just as though they were read from the source file.

A macro definition must begin with the $MACRO macro directive and end with the $END macro directive.

The Location Counter

The Assembler maintains a "location counter" (similar in purpose to the computer's Program Counter) as it assembles code or data.

This location counter is the "address" at which the code or data will be loaded. The location counter value may be either relocatable or absolute. If absolute, then the location counter is the exact memory address at which the code or data must be loaded. If relocatable, then the location counter is a "relative address" that is, relative to the address at which loading starts.

The Assembler begins with its location counter at zero and in relocatable mode. The AORG, RORG, DORG and COMMON Assembler Directives can be used to assign values to the Assembler's location counter. As symbols are encountered in the source code, they are assigned values usually based on the location counter. Along with the value assigned, each symbol has an attribute of "absolute" or "relocatable" depending upon how the symbol was assigned a value.

The value of the Assembler's location counter can be referenced by the special symbol "$".

Expressions

The Assembler allows the use of an arithmetic expression for most operands ("string" operands are an exception). These expressions can contain ordinary symbols, REF symbols, constants and the operators: "+", "-", "*" and "/". Expressions are evaluated in strict left to right order with no operator precedence rules. For example, "2+3*5" evaluates to 25 not to 17 as would a BASIC expression. Parentheses are not allowed in Assembler expressions.

All expressions are evaluated using 16-bit unsigned arithmetic.

When ordinary symbols are used in expressions it is important to keep in mind the relocatability attribute of the symbols. The relocatability rules for expression evaluation are:

```
relocatable + relocatable  is  not allowed
relocatable + absolute     is  relocatable
absolute    + relocatable  is  relocatable
absolute    + absolute     is  absolute

relocatable - relocatable  is  absolute
relocatable - absolute     is  relocatable
absolute    - relocatable  is  relocatable
absolute    - absolute     is  absolute

relocatable * relocatable  is  not allowed
relocatable * absolute     is  not allowed
absolute    * relocatable  is  not allowed
absolute    * absolute     is  absolute

relocatable / relocatable  is  not allowed
relocatable / absolute     is  not allowed
absolute    / relocatable  is  not allowed
absolute    / absolute     is  absolute
```

Note that "constants" are always absolute, and that byte expressions must always be absolute.

Expressions containing REF symbols (which are treated as relocatable symbols) can only have the forms:

```
relocatable + absolute     is  complex relocatable
absolute    + relocatable  is  complex relocatable
```

The "absolute - relocatable" rule above can give rise to a problem which the Assembler does not detect. If you coded a statement like:

```
DATA  -X
```

where X is relocatable, the result (i.e. 0-X) will not be as expected. When assembled, the negative value of X will be assembled into the data word, but when the code is loaded, since the value is relocatable, it will be relocated by the loader. That is, the data value will be:

```
-X + Relocation factor
```

which is not the negative of the address of X.

Constants

The Assembler allows three types of constants. Decimal integers are written in the usual form. Hexadecimal numbers are identified by a leading ">" followed by hex digits 0-9 and A-F. Character constants are identified by enclosing the characters in single quotes "'". The character "'" in a character constant is represented by two single quote marks. Note that character constants can be used in expressions as numbers. The following DATA statements demonstrate the various types of constants.

```
DATA  10          DECIMAL 10
DATA  10*2        DECIMAL 20
DATA  >F          HEXADECIMAL (DECIMAL VALUE 15)
DATA  >000F       SAME AS ABOVE
DATA  'A'         CHARACTER (DECIMAL VALUE 65)
DATA  'A'+1       CHAR (DECIMAL VALUE 65+1=66)
```

Note that character constants are not the same as strings which are defined later.

Definition of Terms

The following terms are used in the descriptions of the Assembler statements in later sections of this manual.

VALUE - means a data value or an expression which evaluates to a data value. The value may be absolute or relocatable.

LABEL - is an ordinary symbol in the label field of a statement. Labels begin in position one of the statement.

NAME - is an ordinary symbol used in an operand field.

DESTINATION - is the result field. For example in A=B, A is the destination. Allowable ways of coding the destination operand is specified in the description of the statement.

SOURCE - is the source field. For example in A=B, B is the source field. Allowable ways of coding the source operand is specified in the description of the statement.

GENERAL ADDRESS - is any one of the forms of addressing allowed by the CPU. These are:

```
@symbol         - Symbolic memory direct
@symbol(Rn)     - Indexed symbolic memory
Rn              - Register direct
*Rn             - Register indirect
*Rn+            - Register indirect with auto-increment
```

COUNT - is an absolute value such as the count of the number of bit positions to be shifted. The range of valid count values is specified in the description of the statement.

DISPLACEMENT - is an absolute value such as the displacement of a CRU bit address from the base CRU address. The range of valid displacement values is specified in the description of the statement.

STRING - is a string of characters. Strings can be coded in one of three ways. First, the characters can be enclosed in single quotation marks (a single quote within the string is represented by two single quotes). Second, the characters can be enclose in double quote marks (a double quote within the string is represented by two double quotes). Third, by a sequence of hexadecimal digits preceded by the hex indicator, ">". The following three strings are all identical:

```
'ASDF'
"ASDF"
>41534446
```

ASSEMBLER DIRECTIVES

AORG - Absolute Origin

```
[label] AORG  value   [comments]
```

The AORG directive assigns an absolute value to the assembler's location counter. All code assembled after the AORG will be in absolute form. The "label" if coded is assigned the new value of the location counter. The "value" expression must contain only previously defined symbols, and the expression must result in an absolute value.
Examples:

```
NEWORG AORG  >F000         ABSOLUTE CODE AT >F000
       AORG  NEWORG+>400   ABSOLUTE CODE AT >F400
```

BES - Block Ending with Symbol

```
[label] BES   value   [comment]
```

The BES directive reserves a block of storage (i.e. a program work area). The "label" if coded is assigned the address of the byte immediately following the reserved block of storage. The number of bytes reserved is specified by the "value" operand. The "value" expression must contain only previously defined symbols, and must be an absolute value.
Examples:

```
WORK   BES   10          WORK AREA OF 10 BYTES
SIZE   EQU   25
       BES   SIZE*2      RESERVE 50 BYTES
```

BSS - Block Starting with Symbol

```
[label] BSS   value   [comment]
```

The BSS directive reserves a block of storage (i.e. a program work area). The "label" if coded is assigned the address of the first byte of the reserved block of storage. The number of bytes reserved is specified by the "value" operand. The "value" expression must contain only previously defined symbols, and must be an absolute value.
Examples:

```
WORK   BSS   10          WORK AREA OF 10 BYTES
SIZE   EQU   25
       BSS   SIZE*2      RESERVE 50 BYTES
```

BYTE - Define a Data Byte

```
[label] BYTE  value,value,...   [comment]
```

The BYTE directive causes constant data values to be assembled into bytes. The "value" expression must be absolute. A number of

byte values can be specified on a single statement by separating the value expressions by commas.
Examples:

```
CON1   BYTE  10          ONE BYTE VALUE OF 10
CON2   BYTE  >20,'A',12  THREE CONSTANTS
SIZE   EQU   25.
NUMBER BYTE  SIZE*2
```

COMMON - Define Relocatable Common Data Area

```
label   COMMON          [comment]
```

The COMMON directive begins the definition of a relocatable common data area. There must be a "label" which is the name of the common area. There is no operands field.

The location counter is set to zero (relocatable) by the COMMON directive. No object code or data is produced for a common area. The assembler operates normally, defining any symbols and producing a listing, except that no object code is produced. Only the name and the size of the common area is output to the object file. The linker or loader must assign the address of the common area.

The linker or loader must inspect all common definitions and select the largest size specified for each name. All object files in a program which define a common name will refer to the same area. If a common name is also defined by a DEF in some object file, then that DEF value will be used to resolve all common references.

The common area or block is terminated by another block definition via the AORG, DORG, RORG, or COMMON directives, or by the END directive.

The common name is treated by the assembler like a REF name. A common block definition is like a relocatable DORG. Any reference in the program to a symbol defined in a common block results in a "complex relocatable value" being produced. That is, a reference to a common defined name is treated as a reference to the common name PLUS an offset.
Example:

```
WORK   COMMON       Define common work area
X      BSS    10
Y      BSS    10
Z      BSS    10
TABLE  COMMON       Define common table
THEAD  BSS    2
       BSS    100
TEND   BSS    2
```

Defines two common areas: WORK of length 30 bytes and TABLE of length 104 bytes.

COPY - Copy Source from File

```
     [label] COPY  string   [comment]
```

The COPY directive causes the file named in the operand field to be read as part of the source file. The name of the file to be read is specified in the usual way in the "string" operand. Note that if the forth character of the file name is an asterisk then the disk number of the source file is substituted. If the entire device part of the file name is specified as a asterisk then the device/directory part of the source file name is substituted. A COPY directive may only occur within the source file and not within a "copy" file.
Examples:

```
            COPY  "DSK1.SRC2" INCLUDE 2ND PART OF SOURCE
     X      COPY  'DSK*.SRC3' SRC3 FILE FROM SOURCE DISK
            COPY  '*.SRC3'    SRC3 FILE FROM SOURCE DISK
```

DATA - Define a Data Word

```
     [label] DATA  value,value,...   [comment]
```

Te DATA directive causes constant data values to be assembled into words. The "value" expression may be absolute or relocatable. A number of word values can be specified on a single statement by separating the value expressions by commas.
Examples:

```
     CON1   DATA  10          ONE WORD VALUE OF 10
     CON2   DATA  >20,'AB',12 THREE CONSTANTS
     SIZE   EQU   25
     NUMBER DATA  SIZE*2      WORD VALUE OF 50
```

DEF - Define External Name

```
     [label] DEF   name,name,...   [comment]
```

The DEF directive specifies that the names in the operand field are "external", that is, they can be referenced by other separately assembled programs. The names listed in the operand field must be defined elsewhere in the program being assembled.
Examples:

```
            DEF   SUB1,SUB2   DEFINE SUBROUTINE ENTRIES
```

DORG - Dummy Origin

```
     [label] DORG  value   [comment]
```

The DORG directive assigns an absolute value to the assembler's location counter. It also directs the assembler not to produce object code for the following code. The assembler operates

normally, defining any symbols which occur and producing a listing if required, except that no object code is written to the object file. The assembler will resume normal operation if an AORG or RORG directive is encountered after the DORG.

The value expression must be absolute. If a label is coded in the label field it will be assigned the new location counter value.
Example:

```
A       DORG  100             BEGIN DUMMY CODE
        DORG  A+1000
        AORG  >F000           RESUME WITH ABS CODE
```

END - End of Assembly

```
[label] END   [name]  [comment]
```

The END directive is the last statement in the program being assembled. A name is specified in the operand field to indicate to the loader or linker where execution of the program is to begin.
Example:

```
        END
```

EQU - Set Symbol Equal to Value

```
[label] EQU   value   [comment]
```

The EQU directive is used to assign a value directly to a symbol. The symbol in the label field is assigned the value and relocatability of the expression in the operand field.
Examples:

```
TEN    EQU   10          SYMBOLIC VALUE 10
TWENTY EQU   TEN*2
X      BSS   2
Y      EQU   X+1         2ND BYTE OF X
```

EQUV - Set Symbol to New Value

```
[label] EQUV  value   [comment]
```

The EQUV directive is used to assign or reassign a value directly to a symbol. The symbol in the label field is assigned the value and relocatability of the expression in the operand field. Note that EQUV is the only way the value of an ordinary symbol can be changed during an assembly. This directive should be used with care as it allows the value of <u>any</u> symbol to be changed.
Examples:

```
TEN    EQUV  10          SYMBOLIC VALUE 10.
TWENTY EQUV  TEN*2
TEN    EQUV  15          CHANGE VALUE OF TEN
Y      EQU   TEN+1       SYMBOLIC VALUE 16
```

EVEN - Location Counter to Even Address

```
[label] EVEN          [comment]
```

The EVEN directive aligns the location counter to a word address (i.e. and even address). If the location counter is odd then it is adjusted up to the next even address. Note that all machine instructions and the DATA directive also align the location counter to a word address. The EVEN directive has no operands.
Example:

```
XXX    EVEN            ALIGN TO WORD BOUNDARY
```

FLOAT - Define Floating Point Value

```
[label] FLOAT f1,f2,...  [comment]
```

The FLOAT directive causes floating point data values to be assembled into the program. The Assembler uses the GPL routines to convert the floating point numbers so that any form acceptable to TI BASIC can be used. A number of floating point numbers can be specified on a single FLOAT statement, separated by commas.

The location counter is aligned on a word boundary prior to assembling the numbers. Each floating point number occupies 8 bytes of memory.
Examples:

```
X      FLOAT 1.0,1E3   1 and 1000
Y      FLOAT -2.8E-2
```

IDT - Identify Object

```
[label] IDT   string  [comment]
```

The IDT directive causes the 1 to 8 character string to be used in the identification field in the object code. If more than one IDT directive is used, the last string specified is used.
Example:

```
           IDT   'JONES'
```

LIST - Resume Assembler Listing

```
[label] LIST          [comment]
```

The LIST directive causes the object listing to be resumed after it has been halted by an UNL directive. The LIST directive has no operands.
Example:

```
           LIST
```

OBJREC - Write Object Record

```
                  {BEFORE}
     [label] OBJREC {AFTER },string  [comment]
                  {NOW   }
```

The OBJREC directive allows arbitrary records to be written into the object file. One important use for this directive could be to add control statements to the object file for use by a linker.

The first operand is a coded value which tells the assembler where in the object file the record is to be written: BEFORE the first object record, AFTER the last object record, or NOW at the current position in the object file.

The OBJREC directives can be placed anywhere in the source program. In particular, the BEFORE text is collected during pass 1 and written, in order, before pass 2 begins, and the AFTER text is collected during pass 2 and written, in order, at the end of pass 2. The NOW text is written as encountered during pass 2 after writing any partial object record that may exist.

The "string" is the text to be placed in the object record. There is a limit to the amount of text that can be saved for either BEFORE or AFTER.
Examples:

```
       OBJREC BEFORE,'LOAD DSK*.SUBS'
       OBJREC AFTER,"ENTRY MAIN"
```

PAGE - Start New Listing Page

```
     [label] PAGE          [comment]
```

The PAGE directive causes the Assembler to start a new page in the listing file.
Example:

```
             PAGE                 START NEW PAGE
```

REF - External Reference

```
     [label] REF   name,name,...  [comment]
```

The REF directive defines the names in the operand field to be references to symbols defined in a separately assembled program. Note that external references may not be used in expressions.
Example:

```
          REF   SUB1,SUB2    DEFINE SUB1 AND SUB2
          BLWP  @SUB2        CALL SUBROUTINE 2
```

RORG - Relocatable Origin

```
[label] RORG  [value]  [comment]
```

The RORG directive assigns a new relocatable value to the assembler's location counter. Even if the value expression is an absolute value, the location counter will be made relocatable. If a label is coded in the label field it will be assigned the new location counter value. If no operand is coded then the location counter is set to the highest relocatable value that has been encountered in the assembly.
Example:

```
A       RORG  100
        RORG  A+1000
        AORG  >F000         ABSOLUTE CODE
* RESUME RELOCATABLE CODE
        RORG
```

STRI - Define ASCII String Constant

```
[label] STRI  string  [comment]
```

The STRI directive assembles a string constant into the program. A string constant has the length of the text as the first byte. This is similar to the TEXT directive except for the leading length byte.
Examples:

```
S1      STRI  'STRING CONSTANT'
S2      STRI  "ANOTHER STRING"
S3      STRI  >52414720534F465457415245
```

TEXT - Define ASCII Text Constant

```
[label] TEXT  string  [comment]
```

The TEXT directive assembles an ASCII character constant into the program.
Examples:

```
T1      TEXT  'ASCII CHARACTERS'
T2      TEXT  "ARE ASSEMBLED INTO"
T3      TEXT  >5448452050524F47414D
```

TITL - Define Listing Title

```
[label] TITL  string  [comment]
```

The TITL directive provides up to 25 characters to be printed in the listing page heading. If TITL is the first statement in the source file then the string will be printed on the first page of the listing. The title can be changed during assembly, the new title string will appear on the next page printed.
Example:

```
          TITL  'NEW PAGE HEADING'
```

UNL - Stop Assembler Listing

```
     [label] UNL           [comment]
```

The UNL directive stops the listing of source and object. The listing can be resumed by the LIST directive.
Example:

```
          UNL
```

ORDINARY STATEMENTS

A - Add Word

```
     [label] A     source,destination   [comment]
```

The source operand value is added to the destination operand value, the sum replacing the destination operand value. Both operands are coded as general addresses. The two 16-bit words may represent either signed or unsigned numbers. The resultant sum is compared to zero to set the L>, A> and EQ status bits. The CA status bit is set when a carry from bit 0 occurs (i.e. unsigned overflow). The OV status bit is set when signed overflow occurs.
Examples:

```
     X      A     @A,@B         B = A + B
            A     @A,R1         REG 1 = A + R1
            A     R0,*R1        ADD R0 TO WORD -> TO BY R1
            A     *R0+,*R1+
     A      DATA  10
     B      DATA  20
```

AB - Add Byte

```
     [label] AB    source,destination   [comment]
```

The source operand value is added to the destination operand value, the sum replacing the destination operand value. Both operands are coded as general addresses. The two 8-bit bytes may represent either signed or unsigned numbers. The resultant sum is compared to zero to set the L>, A> and EQ status bits. The CA status bit is set when a carry from bit 0 occurs (i.e. unsigned overflow). The OV status bit is set when signed overflow occurs. The OP status bit is set when the number of bits in the sum is odd.
Examples:

```
     X      AB    @A,@B         B = A + B
            AB    @A,R1         REG 1 = A + R1
     Y      AB    R0,*R1        ADD R0 TO BYTE -> TO BY R1
            AB    *R0+,*R1+
     A      BYTE  10
     B      BYTE  20
```

ABS - Absolute Value

```
[label] ABS   destination          [comment]
```

The destination operand value, which is considered to be a signed number, is made positive. If the destination value is already positive, no change takes place. The original value is compared to zero to set the L>, A> and EQ status bits. NOTE: the original value is used. The OV status bit will be set if the original value is >8000.
Examples:

```
X       ABS   @A            ABSOLUTE VALUE OF A
        ABS   R1            ABSOLUTE VALUE OF REG 1
Y       ABS   *R1           ABS OF WORD POINTED TO BY R1
        ABS   *R0+
A       BSS   2
```

AI - Add Immediate

```
[label] AI   destination,value    [comment].
```

The 16-bit immediate data value is added to the destination operand value, the sum replacing the destination operand value. The destination operand must be specified as a workspace register. The two 16-bit words may represent either signed or unsigned numbers. The resultant sum is compared to zero to set the L>, A> and EQ status bits. The CA status bit is set when a carry from bit 0 occurs (i.e. unsigned overflow). The OV status bit is set when signed overflow occurs.
Examples:

```
X       AI    R1,20         R1 = R1 + 20
        AI    R0,-30        REG 0 = REG 0 - 30
        AI    R15,A
A       EQU   10
```

ANDI - And Immediate

```
[label] ANDI destination,value    [comment]
```

The logical AND of the 16-bit immediate data value and the destination operand value is performed. The result replaces the destination operand value. The destination operand must be specified as a workspace register. The result is compared to zero to set the L>, A> and EQ status bits.
Examples:

```
X       ANDI  R1,>F000     ISOLATE 1ST NIBBLE OF R1
        ANDI  R0,MASK
MASK    EQU   000F
```

B - Branch

```
[label] B     destination    [comment]
```

Branch to, or continue execution at, the destination address. The destination is specified as a general address.
Examples:

```
X       B     @A            CONTINUE AT LABEL A
        B     *R1           CONTINUE AT ADDRESS IN REG 1
A       EVEN
```

BL - Branch and Link

```
[label] BL    destination    [comment]
```

Branch to, or continue execution at, the destination address saving the current address in register 11. The destination is specified as a general address. The program can continue execution at the instruction following the BL by branching to the address saved in register 11.
Examples:

```
X       BL    @A            BRANCH AND LINK TO ROUTINE A
        BL    R1            CALL ROUTINE AT ADDR IN R1
A       EVEN
```

BLWP - Branch and Load Workspace Pointer

```
[label] BLWP  destination    [comment]
```

The two words at the destination address are used to load the Workspace Pointer and Program Counter, thus defining a new register set and continuing execution at a new address. The old workspace pointer is saved in register 13 of the new workspace. The old program counter (the return address) is saved in register 14 of the new workspace. The old status register is saved in register 15 of the new workspace. The destination is specified as a general address. The program can restore the old workspace pointer, the old status register, and can continue execution at the instruction following the BLWP by using the RTWP instruction.
Examples:

```
X       BLWP  @A            CALL SUBROUTINE A
        BLWP  *R1           CALL SUBRTN AT ADDR IN R1
A       DATA  NEWWSP,NEWPC  SUBROUTINE "A" BLWP VECTOR
```

CLR - Clear

```
[label] CLR   destination    [comment]
```

The destination operand word is set to zero. The destination operand is specified as a general address. No status bits are affected by this instruction.
Examples:

```
X       CLR   @A            ZERO VALUE IN A
        CLR   R1            ZERO REGISTER 1
Y       CLR   *R1           ZERO WORD POINTED TO BY R1
        CLR   *R0+
A       BSS   2
```

C - Compare Word

```
[label] C    source,destination   [comment]
```

The source operand value is compared to the destination operand value. Both operands are coded as general addresses. Neither operand is changed. The two 16-bit words may represent either signed or unsigned numbers. The L>, A> and EQ status bits are set to reflect the result of the compare.
Examples:

```
X       C     @A,@B         COMPARE A TO B
        C     @A,R1         COMPARE A TO VALUE IN R1
        C     R0,*R1        COMP R0 TO WORD -> TO BY R1
        C     *R0+,*R1+
A       DATA  10
B       DATA  20
```

CB - Compare Byte

```
[label] CB    source,destination   [comment]
```

The source operand value is compared to the destination operand value. Both operands are coded as general addresses. Neither operand is changed. The two 8-bit bytes may represent either signed or unsigned numbers. The L>, A> and EQ status bits are set to reflect the result of the compare. The OP status bit is set when the number of bits in the source operand is odd.
Examples:

```
X       CB    @A,@B         COMPARE A TO B
        CB    @A,R1         COMPARE A TO 1ST BYTE OF R1
Y       CB    R0,*R1        COMP R0 TO BYTE -> TO BY R1
        CB    *R0+,*R1+
A       BYTE  10
B       BYTE  20
```

CI - Compare Immediate

```
[label] CI   source,value    [comment]
```

The 16-bit source value is compared to the immediate data value. The source operand must be specified as a workspace register. Neither operand is changed. The L>, A> and EQ status bits are set to reflect the result of the compare.
Examples:

```
X       CI    R1,20         COMPARE VALUE IN R1 TO 20
```

```
        CI    R0,-30       COMPARE VALUE IN R0 TO -30
        CI    R15,A        COMPARE VALUE IN R15 TO 10
        CI    R15,A+25     COMPARE VALUE IN R15 TO 35
A       EQU   10
```

COC - Compare Ones Corresponding

```
[label] COC  source,destination   [comment]
```

The 16-bit source value is compared to the 16-bit value in the destination register. If all bits in the source value that are ones are also ones in the destination then the EQ status is set, otherwise the EQ status bit is reset. The source operand is specified as a general address. The destination operand is specified as a workspace register.

The COC instruction tests single or multiple bits in a workspace register.
Examples:

```
X       COC   R1,R2         COMPARE ONE BITS R1 TO R2
        JEQ   ON            JUMP ALL BITS ON
        COC   @Y,R3         TEST FOR ONE BITS IN R3
        JNE   OFF           JUMP BITS NOT ALL ON
```

CZC - Compare Zeros Corresponding

```
[label] CZC  source,destination   [comment]
```

The 16-bit source value is compared to the 16-bit value in the destination register. If all bits in the source value that are ones are zeros in the destination then the EQ status is set, otherwise the EQ status bit is reset. The source operand is specified as a general address. The destination operand is specified as a workspace register.
Examples:

```
X       CZC   R1,R2         TEST FOR ZERO BITS IN R2
        JEQ   ZEROS         JUMP ALL BITS ZEROS
        CZC   @Y,R3         COMPARE ONE BITS OF Y TO R3
        JNE   MIXED         JUMP SOME BITS NOT ZERO
```

DEC - Decrement

```
[label] DEC   destination    [comment]
```

The 16-bit destination operand value is decremented by one. The resultant value is compared to zero to set the L>, A> and EQ status bits. The CA status bit is set when a carry from bit 0 occurs (i.e. unsigned overflow). The OV status bit is set when signed overflow occurs.
Examples:

```
X       DEC   @A            A = A - 1
        DEC   R1            SUBTRACT 1 FROM REG 1
```

```
Y       DEC   *R1          DECREMENT WORD AT ADDR IN R1
        DEC   *R0+
A       BSS   2
```

DECT - Decrement by Two

```
[label] DECT  destination    [comment]
```

The 16-bit destination operand value is decremented by two. The resultant value is compared to zero to set the L>, A> and EQ status bits. The CA status bit is set when a carry from bit 0 occurs (i.e. unsigned overflow). The OV status bit is set when signed overflow occurs.
Examples:

```
X       DECT  @A            A = A - 2
        DECT  R1            SUBTRACT 2 FROM REG 1
Y       DECT  *R1           WORD AT ADDR IN R1 LESS 2
        DECT  *R0+
A       BSS   2
```

DIV - Divide

```
[label] DIV   source,destination   [comment]
```

The 32-bit destination operand value is divided by the 16-bit source operand value using unsigned integer rules. The source operand is specified as a general address. The destination operand must be specified as a workspace register. The quotient and remainder replace the 2-word destination. The quotient is placed in the specified destination register, and the remainder is placed in the specified register + 1. Note if the destination register is specified as register 15, then the destination operand extends into the word of memory following the workspace. The OV status bit is set if the resultant quotient cannot be expressed in 16 bits.
Examples:

```
X       DIV   @A,R0         DIVIDE A INTO (R0,R1).
        DIV   R1,R2         DIVIDE R1 INTO (R2,R3)
Y       DIV   *R1,R3        DIVIDE VALUE -> TO BY R1
*                           INTO (R3,R4)
        DIV   *R0+,R8
A       BSS   2
```

INC - Increment

```
[label] INC   destination    [comment]
```

The 16-bit destination operand value is incremented by one. The resultant value is compared to zero to set the L>, A> and EQ status bits. The CA status bit is set when a carry from bit 0 occurs (i.e. unsigned overflow). The OV status bit is set when signed overflow occurs.
Examples:

```
X       INC   @A           A = A + 1
        INC   R1           ADD 1 TO REG 1
Y       INC   *R1          INCREMENT WORD -> TO BY R1
        INC   *R0+
A       BSS   2
```

INCT - Increment by Two

```
[label] INCT  destination   [comment]
```

The 16-bit destination operand value is incremented by two. The resultant value is compared to zero to set the L>, A> and EQ status bits. The CA status bit is set when a carry from bit 0 occurs (i.e. unsigned overflow). The OV status bit is set when signed overflow occurs.
Examples:

```
X       INCT  @A           A = A + 2
        INCT  R1           ADD 2 TO REG 1
Y       INCT  *R1          WORD POINTED TO BY R1 + 2
        INCT  *R0+
A       BSS   2
```

INV - Invert

```
[label] INV   destination   [comment]
```

The 16-bit destination operand value is replaced by its one's complement. The destination operand is specified as a general address. The one's complement is obtained by changing each zero bit to a one bit, and each one bit to a zero bit. The result value is compared to zero to set the L>, A> and EQ status bits.
Examples:

```
X       INV   @A           INVERT BITS OF A
        INV   R1           INVERT BITS OF REG 1
Y       INV   *R1          INVERT BITS OF WORD -> BY R1
        INV   *R0+
A       BSS   2
```

JEQ - Jump if Equal

```
[label] JEQ   destination   [comment]
```

If the EQ status bit is set, program execution continues at the destination label. The destination is specified as a label. The label must be within the range of -127 to +128 words of the address of the JEQ instruction. JEQ can be used to test the result of either signed or unsigned arithmetic or compares.
Examples:

```
X       JEQ   A          JUMP IF EQUAL
        JEQ   B
```

JGT - Jump if Greater Than

```
     [label] JGT   destination    [comment]
```

If the A> status bit is set, program execution continues at the destination label. The destination is specified as a label. The label must be within the range of -127 to +128 words of the address of the JGT instruction. JGT can be used to test the result of signed arithmetic or compares.
Examples:

```
     X      JGT   A           JUMP IF GREATER THAN
            JGT   B
```

JH - Jump if Logical High

```
     [label] JH    destination    [comment]
```

If the L> status bit is set, program execution continues at the destination label. The destination is specified as a label. The label must be within the range of -127 to +128 words of the address of the JH instruction. JH can be used to test the result of unsigned arithmetic or compares.
Examples:

```
     X      JH    A           JUMP IF LOGICALLY HIGH
            JH    B
```

JHE - Jump if Logical High or Equal

```
     [label] JHE   destination    [comment]
```

If either the L> status bit or the EQ status bit is set, program execution continues at the destination label. The destination is specified as a label. The label must be within the range of -127 to +128 words of the address of the JHE instruction. JHE can be used to test the result of unsigned arithmetic or compares.
Examples:

```
     X      JHE   A           JUMP IF LOG HIGH OR EQUAL
            JHE   B
```

JL - Jump if Logical Low

```
     [label] JL    destination    [comment]
```

If neither the EQ status bit nor the L> status bit is set, program execution continues at the destination label. The destination is specified as a label. The label must be within the range of -127 to +128 words of the address of the JL instruction. JL can be used to test the result of unsigned arithmetic or compares.
Examples:

```
     X      JL    A           JUMP IF LOGICAL LOW
            JL    B
```

JLE - Jump if Logical Low or Equal

```
[label] JLE   destination    [comment]
```

If the L> status bit is not set or if the EQ status bit is set, program execution continues at the destination label. The destination is specified as a label. The label must be within the range of -127 to +128 words of the address of the JLE instruction. JLE can be used to test the result of unsigned arithmetic or compares.
Examples:

```
X       JLE   A          JUMP IF LOG LOW OR EQUAL
        JLE   B
```

JLT - Jump if Less Than

```
[label] JLT   destination    [comment]
```

If neither the A> status bit nor the EQ status bit is set, program execution continues at the destination label. The destination is specified as a label. The label must be within the range of -127 to +128 words of the address of the JLT instruction. JLT can be used to test the result of signed arithmetic or compares.
Examples:

```
X       JLT   A          JUMP IF ARITH LESS THAN
        JLT   B
```

JMP - Jump Unconditionally

```
[label] JMP   destination    [comment]
```

The JMP causes program execution to continue at the destination label. The destination is specified as a label. The label must be within the range of -127 to +128 words of the address of the JMP instruction.
Examples:

```
X       JMP   A          CONTINUE AT A
        JMP   B
```

JNC - Jump if No Carry

```
[label] JNC   destination    [comment]
```

If the CA status bit is not set, program execution continues at the destination label. The destination is specified as a label. The label must be within the range of -127 to +128 words of the address of the JNC instruction.
Examples:

```
X       JNC   A          JUMP IF NO CARRY
        JNC   B
```

JNE - Jump if Not Equal

```
[label] JNE   destination    [comment]
```

If the EQ status bit is not set, program execution continues at the destination label. The destination is specified as a label. The label must be within the range of -127 to +128 words of the address of the JNE instruction. JNE can be used to test the result of signed or unsigned arithmetic or compares.
Examples:

```
X       JNE   A           JUMP NOT EQUAL TO ZERO
        JNE   B
```

JNO - Jump if No Overflow

```
[label] JNO   destination    [comment]
```

If the OV status bit is not set, program execution continues at the destination label. The destination is specified as a label. The label must be within the range of -127 to +128 words of the address of the JNO instruction.
Examples:

```
X       JNO   A           JUMP NO OVERFLOW
        JNO   B
```

JOC - Jump On Carry

```
[label] JOC   destination    [comment]
```

If the CA status bit is set, program execution continues at the destination label. The destination is specified as a label. The label must be within the range of -127 to +128 words of the address of the JOC instruction.
Examples:

```
X       JOC   A           JUMP IF CARRY BIT ON
        JOC   B
```

JOP - Jump if Odd Parity

```
[label] JOP   destination    [comment]
```

If the OP status bit is set, program execution continues at the destination label. The destination is specified as a label. The label must be within the range of -127 to +128 words of the address of the JOP instruction.
Examples:

```
X       JOP   A           JUMP BYTE HAS ODD # OF BITS
        JOP   B
```

LDCR - Load CRU

```
     [label] LDCR  source,count    [comment]
```

Transfer the number of bits specified by the count operand from the source operand to the CRU. The source operand is specified as a general address. The count operand is specified as a value in the range 0 to 15. The transfer begins with the least significant bit of the source operand value. The CRU address is contained in workspace register 12. When the count operand is 0, 16 bits are transferred. If the number of bits to transfer is 8 or less then the source operand is a byte address, otherwise it is a word address. The source operand value is compared to zero to set the L>, A> and EQ status bits. If the number of bits to transfer is 8 or less then the source operand byte is tested to set the OP status bit.
Examples:

```
     X      LDCR  @A,5         SEND LOW 5 BITS OF BYTE A
            LDCR  R1,0         TRANSFER 16 BITS OF REG 1
```

LI - Load Immediate

```
     [label] LI   destination,value    [comment]
```

The 16-bit immediate data value is loaded into the destination operand value. The destination operand must be specified as a workspace register. The immediate data value is compared to zero to set the L>, A> and EQ status bits.
Examples:

```
     X      LI    R1,20        R1 = 20.
            LI    R0,-30       REG 1 = -30
            LI    R15,A        R15 = 10
     A      EQU   10
```

LIMI - Load Interrupt Mask Immediate

```
     [label] LIMI mask          [comment]
```

The least significant 4 bits of the immediate mask value are loaded into the interrupt mask of the Status Register. The remainder of the Status Register is not affected.
Examples:

```
     X      LIMI  0            DISABLE INTERRUPTS.
            LIMI  2            ENABLE INTERRUPTS 1 AND 2
```

LWPI - Load Workspace Pointer Immediate

```
     [label] LWPI value         [comment]
```

The immediate value is loaded into the Workspace Pointer Register. No status bits are affected by this instruction.
Examples:

```
     X      LWPI  >F000        SWITCH TO WORKSPACE AT >F000
```

```
        LWPI  A           USE WORKSPACE A
A       BSS   32
```

MOV - Move Word

```
[label] MOV   source,destination     [comment]
```

The 16-bit source operand value is moved to the destination operand. Both operands are coded as general addresses. The value moved is compared to zero to set the L>, A> and EQ status bits.
Examples:

```
X       MOV   @A,@B       B = A
        MOV   @A,R1       MOVE A TO REGISTER 1
        MOV   R0,*R1      MOVE R0 TO AREA -> TO BY R1
        MOV   *R0+,*R1+
A       DATA  10
B       DATA  20
```

MOVB - Move Byte

```
[label] MOVB  source,destination     [comment]
```

The 8-bit source operand value is moved to the destination operand. Both operands are coded as general addresses. The byte moved is compared to zero to set the L>, A> and EQ status bits. The OP status bit is set when the number of bits in the byte is odd.
Examples:

```
X       MOVB  @A,@B       B = A
        MOVB  @A,R1       MOVE BYTE FROM A TO REG 1
Y       MOVB  R0,*R1      MOVE BYTE IN R0 TO ADD IN R1
        MOVB  *R0+,*R1+
A       BYTE  10
B       BYTE  20
```

MPY - Multiply

```
[label] MPY   source,destination     [comment]
```

The 16-bit source operand value is multiplied by the 16-bit destination operand value using unsigned integer rules. The source operand is specified as a general address. The destination operand must be specified as a workspace register. The result of the multiplication is a 32-bit value that replaces the value in the specified destination register and the value in the specified register + 1. Note if the destination register is specified as register 15, then the destination operand extends into the word of memory following the workspace. No status bits are affected by this instruction.
Examples:

```
X       MPY   @A,R0       (R0,R1)= A * R0
        MPY   R1,R2       (R2,R3)= R1 * R2
```

```
Y       MPY   *R1,R3       (R3,R4)=(WORD -> BY R1) * R3
        MPY   *R0+,R8
A       BSS   2
```

NEG - Negate

```
[label] NEG   destination     [comment]
```

The destination operand value, which is considered to be a signed number, is replaced by its two's complement. If the number is positive it is made negative. If the number is negative it is made positive. The result is compared to zero to set the L>, A> and EQ status bits. The OV status bit will be set if the original value is >8000.
Examples:

```
X       NEG   @A            CHANGE SIGN OF VALUE AT A
        NEG   R1            NEGATE VALUE IN REG 1
Y       NEG   *R1           NEG OF WORD -> TO BY R1
        NEG   *R0+
A       BSS   2
```

NOP - No Operation

```
[label] NOP                  [comment]
```

This instruction is exactly like a "JMP $+2". NOP has no operands. No status bits are affected.
Examples:

```
X       NOP                 NO OPERATION
        JMP   $+2
```

ORI - Or Immediate

```
[label] ORI  destination,data   [comment]
```

The logical OR of the 16-bit immediate data value and the destination operand value is performed. The result replaces the destination operand value. The destination operand must be specified as a workspace register number. The result is compared to zero to set the L>, A> and EQ status bits.
Examples:

```
X       ORI   R1,>F000     MAKE 1ST NIBBLE OF R1 ONES
        ORI   R0,MASK
MASK    EQU   >000F
```

RT - Return

```
[label] RT                  [comment]
```

This instruction is exactly like a "B *R11". RT has no operands. No status bits are affected. This instruction is usually used to return from a routine invoked via the BL instruction.

Example:

```
     X      RT                  RETURN TO MAINLINE CODE
```

RTWP - Return With Workspace Pointer

```
     [label] RTWP               [comment]
```

This instruction performs the following operations: the Status Register is loaded from register 15, the Program Counter Register is loaded from register 14 and the Workspace Pointer Register is loaded from register 13. RTWP has no operands. This instruction is usually used to return from a routine invoked via the BLWP instruction.
Examples:

```
     X      RTWP               RETURN TO MAINLINE CODE
```

S - Subtract Word

```
     [label] S     source,destination     [comment]
```

The source operand value is subtracted from the destination operand value, the result replacing the destination operand value. Both operands are coded as general addresses. The two 16-bit words may represent either signed or unsigned numbers. The result is compared to zero to set the L>, A> and EQ status bits. The CA status bit is set when a carry from bit 0 occurs (i.e. unsigned overflow). The OV status bit is set when signed overflow occurs.
Examples:

```
     X      S     @A,@B       B = A - B
            S     @A,R1       REG 1 = A - R1
            S     R0,*R1      SUB R0 FROM WORD -> TO BY R1
            S     *R0+,*R1+
     A      DATA  10
     B      DATA  20
```

SB - Subtract Byte

```
     [label] SB    source,destination     [comment]
```

The source operand value is subtracted from the destination operand value, the result replacing the destination operand value. Both operands are coded as general addresses. The two 8-bit bytes may represent either signed or unsigned numbers. The result is compared to zero to set the L>, A> and EQ status bits. The CA status bit is set when a carry from bit 0 occurs (i.e. unsigned overflow). The OV status bit is set when signed overflow occurs. The OP status bit is set when the number of bits in the result is odd.
Examples:

```
     X      SB    @A,@B       B = A - B
            SB    @A,R1       REG 1 = A - R1
```

```
Y       SB    R0,*R1       SUB R0 FROM BYTE -> TO BY R1
        SB    *R0+,*R1+
A       BYTE  10
B       BYTE  20
```

SBO - Set Bit One

```
[label] SBO   displacement    [comment]
```

Sets the selected CRU bit to one. The CRU bit address is the sum of the displacement operand and the CRU base address. The CRU base address is contained in bits 0 to 14 of workspace register 12. The displacement operand is a number in the range -128 to +127. No status bits are affected by this instruction.
Examples:

```
X       SBO   5             SET CRU RELATIVE BIT 5
        SBO   -2            SET CRT RELATIVE BIT -2
```

SBZ - Set Bit Zero

```
[label] SBZ   displacement    [comment]
```

Sets the selected CRU bit to zero. The CRU bit address is the sum of the displacement operand and the CRU base address. The CRU base address is contained in bits 0 to 14 of workspace register 12. The displacement operand is a number in the range -128 to +127. No status bits are affected by this instruction.
Examples:

```
X       SBZ   5             RESET CRU RELATIVE BIT 5
        SBZ   -2            RESET CRT RELATIVE BIT -2
```

SETO - Set to Ones

```
[label] SETO  destination    [comment]
```

The destination operand word is set to all one bits (>FFFF). The destination operand is specified as a general address. No status bits are affected by this instruction.
Examples:

```
X       SETO  @A            A = >FFFF
        SETO  R1            REGISTER 1 = -1
Y       SETO  *R1           WORD AT ADDR IN R1 = >FFFF
        SETO  *R0+
A       BSS   2
```

SLA - Shift Left Arithmetic

```
[label] SLA   destination,count    [comment]
```

Shifts the 16-bit destination register left by the number of bits specified in the count operand. The count operand is in the range 0 to 15. If the count is zero, the shift count is specified in the

4 least significant bits of workspace register 0. If the least significant 4 bits of register 0 are also zero then the shift is 16 bits. The result is compared to zero to set the L>, A> and EQ status bits. The CA status bit is set to the last bit shifted out of the register. The OV status bit is set if the sign bit (bit 0) changes from 0 to 1 or from 1 to 0 during the shift operation.
Examples:

```
X       SLA   R1,4        SHIFT R1 4 BITS LEFT
        SLA   R1,0        SHIFT LEFT BY # BITS IN R0
        SLA   R2,1        MULTIPLY R2 BY 2
```

SOC - Set Ones Corresponding Word

```
[label] SOC   source,destination    [comment]
```

The source operand value is ORed into the destination operand value, the result replacing the destination operand value. Both operands are coded as general addresses, and are 16-bit words. The result is compared to zero to set the L>, A> and EQ status bits.
Examples:

```
X       SOC   @A,@B        B = A OR B
        SOC   @A,R1        REG 1 = A OR R1
        SOC   R0,*R1       OR R0 INTO WORD AT ADD IN R1
        SOC   *R0+,*R1+
A       DATA  10
B       DATA  20
```

SOCB - Set Ones Corresponding Byte

```
[label] SOCB  source,destination    [comment]
```

The source operand value is ORed into the destination operand value, the result replacing the destination operand value. Both operands are coded as general addresses, and are 8-bit bytes. The result is compared to zero to set the L>, A> and EQ status bits. The OP status bit is set when the number of bits in the result is odd.
Examples:

```
X       SOCB  @A,@B        B = A OR B
        SOCB  @A,R1        REG 1 = A OR R1
Y       SOCB  R0,*R1       OR R0 INTO BYTE -> TO BY R1
        SOCB  *R0+,*R1+
A       BYTE  10
B       BYTE  20
```

SRA - Shift Right Arithmetic

```
[label] SRA   destination,count    [comment]
```

Shifts the 16-bit destination register right by the number of bits specified in the count operand. The count operand is in the range

0 to 15. If the count is zero, the shift count is specified in the 4 least significant bits of workspace register 0. If the least significant 4 bits of register 0 are also zero then the shift is 16 bits. Vacated bits on the left are filled with the original bit 0. The result is compared to zero to set the L>, A> and EQ status bits. The CA status bit is set to the last bit shifted out of the register.
Examples:

```
     X      SRA   R1,4        SHIFT R1 4 BITS RIGHT
            SRA   R1,0        SHIFT RIGHT BY # BITS IN R0
            SRA   R1,1        DIVIDE R1 BY 2
```

SRC - Shift Right Circular

```
     [label] SRC   destination,count    [comment]
```

Shifts the 16-bit destination register right by the number of bits specified in the count operand. The count operand is in the range 0 to 15. If the count is zero, the shift count is specified in the 4 least significant bits of workspace register 0. If the least significant 4 bits of register 0 are also zero then the shift is 16 bits. Each bit shifted out of the right end of the register is shifted into the left end. The result is compared to zero to set the L>, A> and EQ status bits. The CA status bit is set to the last bit shifted out of the register.
Examples:

```
     X      SRC   R1,4        SHIFT R1 4 BITS RIGHT
            SRC   R1,0        SHIFT RIGHT BY # BITS IN R0
```

SRL - Shift Right Logical

```
     [label] SRL   destination,count    [comment]
```

Shifts the 16-bit destination register right by the number of bits specified in the count operand. The count operand is in the range 0 to 15. If the count is zero, the shift count is specified in the 4 least significant bits of workspace register 0. If the least significant 4 bits of register 0 are also zero then the shift is 16 bits. Vacated bits on the left are filled with zero bits. The result is compared to zero to set the L>, A> and EQ status bits. The CA status bit is set to the last bit shifted out of the register.
Examples:

```
     X      SRL   R1,4        SHIFT R1 4 BITS RIGHT
            SRL   R1,0        SHIFT RIGHT BY # BITS IN R0
```

STCR - Store CRU

```
     [label] STCR  destination,count   [comment]
```

Transfer the number of bits specified by the count operand from the CRU to the destination operand. The destination operand is

specified as a general address. The count operand is specified as a value in the range 0 to 15. The transfer begins into the least significant bit of the destination operand value. The CRU address is contained in workspace register 12. When the count operand is 0, 16 bits are transferred. If the number of bits to transfer is 8 or less then the destination operand is a byte address, otherwise it is a word address. The destination operand value is compared to zero to set the L>, A> and EQ status bits. If the number of bits to transfer is 8 or less then the destination operand byte is tested to set the OP status bit.
Examples:

```
     X      STCR  @A,5        GET CRU BITS INTO THE.
     *                        LOW 5 BITS OF BYTE A.
            STCR  R1,0        GET 16 BITS INTO REGISTER 1
```

STST - Store Status Register

```
     [label] STST destination    [comment]
```

The Status Register is stored into the destination operand. The destination operand is specified as a workspace register. No status bits are affected by this instruction.
Examples:

```
     X      STST  R0           SAVE STATUS IN R0
            STST  R15
```

STWP - Store Workspace Pointer

```
     [label] STWP destination    [comment]
```

The Workspace Pointer Register is stored into the destination operand. The destination operand is specified as a workspace register. No status bits are affected by this instruction.
Examples:

```
     X      STWP  R0           SAVE WP IN R0
            STWP  R15
```

SWPB - Swap Bytes

```
     [label] SWPB  destination    [comment]
```

The two bytes in the destination operand word are interchanged. The destination operand is specified as a general address. No status bits are affected by this instruction.
Examples:

```
     X      SWPB  @A           INTERCHANGE BYTES OF A.
            SWPB  R1           INTERCHANGE BYTES IN R1.
     Y      SWPB  *R1          SWAP OF BYTES -> TO BY R1
            SWPB  *R0+
     A      BSS   2
```

SZC - Set Zeros Corresponding Word

```
     [label] SZC   source,destination    [comment]
```

The source operand value is examined and for each 1 bit found, the corresponding bit in the destination operand value is set to zero. Both operands are coded as general addresses, and are 16-bit words. The result is compared to zero to set the L>, A> and EQ status bits.
Examples:

```
     X      SZC   @A,@B       ZERO LAST BIT OF B
            SZC   @C,R1       ZERO 1ST BIT OF R1
            SZC   R0,*R1
            SZC   *R0+,*R1+
     A      DATA  >0001
     B      DATA  20
     C      DATA  >8000
```

SZCB - Set Zeros Corresponding Byte

```
     [label] SZCB  source,destination    [comment]
```

The source operand value is examined and for each 1 bit found, the corresponding bit in the destination operand value is set to zero. Both operands are coded as general addresses, and are 8-bit bytes. The result is compared to zero to set the L>, A> and EQ status bits. The OP status bit is set when the number of bits in the result is odd.
Examples:

```
     X      SZCB  @A,@B       ZERO LAST BIT OF BYTE B
            SZCB  @A,R1       ZERO 1ST BIT OF R1
     Y      SZCB  R0,*R1
            SZCB  *R0+,*R1+
     A      BYTE  >01
     B      BYTE  20
     C      BYTE  >80
```

TB - Test CRU Bit

```
     [label] TB    displacement    [comment]
```

Tests the selected CRU bit. The CRU bit address is the sum of the CRU base address and the signed displacement in the operand field. The CRU base address is contained in bits 0 to 14 of workspace register 12. The displacement operand is a number in the range -128 to +127. The EQ status bit is set if the CRU bit was set.
Examples:

```
            LI    R12,>1100   CRU BASE ADDRESS
     X      TB    4           TEST BIT AT >1108
```

X - Execute

```
     [label] X     destination    [comment]
```

The machine instruction at the destination address is executed. The destination address is specified as a general address. When the instruction at the destination address is not a single word instruction then the extra words of the instruction are fetched from memory following the X instruction. If the executed instruction is a JMP then the jump is taken relative to the PC which is positioned just after the X instruction. No status bits are set by the X instruction, but the subject instruction sets the status bits as normal for that instruction.
Examples:

```
     X      X     @A           EXECUTE INST AT A
            X     R15          EXECUTE INST IN R15
```

XOP - Extended Operation

```
     [label] XOP   source,value    [comment]
```

The XOP instruction causes a context switch, similar to BLWP, to the transfer vector selected by the "value" operand. The "value" operand is a number in the range 0 to 15. The transfer vector for value 0 is at location >0040, for value 1 is at location >0044, and etc. The transfer vector is a workspace pointer address and a program counter value. During execution of the XOP instruction, the current workspace pointer is saved in register 13 of the new workspace, the current program counter is saved in register 14 of the new workspace, the current status register is saved in register 15 of the new workspace, and the address specified by the "source" operand is stored in register 11 of the new workspace. The X status bit is set indicating the XOP instruction has been executed. The routine called by the XOP instruction can resume execution at the code following the XOP by using the RTWP to restore the WP, PC and ST registers.

The XOP instruction is usually used to request a service from the operating system.
Examples:

```
     X      XOP   @A,0         TRANSFER TO XOP 0 ROUTINE
            XOP   R15,1        TRANSFER TO XOP 15 CODE
```

XOR - Exclusive OR

```
[label] XOR   source,destination    [comment]
```

The bit-wise exclusive OR of the 16-bit source and destination operands replaces the destination operand. The source operand is coded as a general address. The destination operand is coded as a workspace register. The exclusive OR of two bits is defined as 0 if both bits are 0 or if both bits are 1; otherwise the result is 1. The result of the exclusive OR is compared to zero to set the L>, A> and EQ status bits.
Examples:

```
X       XOR   @A,R0          XOR OF A AND R0
        XOR   R15,R2         R2= R15 XOR R2
A       DATA  9
```

MACRO DIRECTIVES

Macro directives are used to define macros or to define new instruction operation codes. Macro definitions and operation code definitions may be placed in the source file or in the macro library file. Macro directive statements have a different form than the other Assembler statements. There is no label field on macro directives and the directive operator must begin in position one of the statement. All macro directive operators begin with a dollar sign so that all macro directive statements begin with a dollar sign in position one. The macro directives are described in the following sections.

$END - End of Macro Definition

```
$END                  [comment]
```

A macro definition must end with the $END directive. There are no operands on this directive.

Example. A macro definition appears as follows:

```
$MACRO BNE
    .
    .
    .
$END
```

$ERROR - Issue Error Message

```
$ERROR string         [comment]
```

This directive causes an assembler error message to be printed. The operand field contains the message string to be inserted into the standard assembler error message line. Any macro symbols in the string are replaced by their values. The maximum length of an assembler error message is 20 characters. This directive is useful for issuing diagnostics when the parameters to a macro are not correct.
Examples:

```
$ERROR '&P1 OPERAND INVALID'
$ERROR 'INCORRECT VALUE'
```

$EXIT - Exit from Macro

```
$EXIT                 [comment]
```

The $EXIT macro indicates the end of macro generation. Note that the $END directive which defines the physical end of a macro

definition also indicates the end of macro generation. The $EXIT directive has no operands.

$GOTO - Branch Within Macro

```
$GOTO  label         [comment]
```

This directive causes a GOTO within a macro definition. The operand field contains the target label which must appear on a $LABEL directive. The operand field is scanned and any macro symbols are replaced by their value before the search for the label is begun.
Examples:

```
$GOTO XYZ
$GOTO &L2
$GOTO X&G3
$GOTO &P1(1.2)
```

$IF - Conditional Branch Within Macro

```
$IF  expr1,relop,expr2,label   [comment]
```

This directive causes a conditional branch within the macro definition. The two expressions are evaluated in the same way as the expression on a $SET directive. The two results are then compared as character strings. If the relation specified by the relational operator, "relop", is true then a $GOTO is executed to the label specified as the fourth operand.

The relational operators are:

```
EQ - equal
NE - not equal
GT - greater than
GE - greater than or equal
LT - less than
LE - less than or equal
```

If the two strings being compared are different lengths and are the same up to the length of the shorter, then the shorter string is less than the longer. For example, 'XYZ' is less than 'XYZA'.
Examples:

```
$IF '&P1',EQ,'XYZ',ISXYZ
$IF &P2,LT,3,L21
$IF '&P3',NE,'&G1&G2',NEW
$IF '&P4',GE,'A123',&G5
 ...
$LABEL ISXYZ
 ...
$LABEL L21
 ...
$LABEL NEW
```

$LABEL - Define Macro Label

```
$LABEL label          [comment]
```

This directive defines a label which may be the target of a $GOTO or $IF directive. The operand field contains the label. The label must be 1 to 6 characters, the first of which is a letter. No macro symbols are allowed.
Examples:

```
$LABEL XYZ.
$LABEL A12345
```

$MACRO - Begin Macro Definition

```
$MACRO name           [comment]
```

A macro definition must begin with the $MACRO directive. The "name" specified is the macro name and is used as an operation code to invoke the macro. The name must be from 1 to 6 characters the first of which must be a letter. Macro names must be different from any predefined instruction operation code or assembler directive operation code.
Examples:

```
$MACRO BNE
$MACRO TEST23
```

$OPCODE - Define Operation Code

```
$OPCODE name,value,type  [comment]
```

This directive defines a new operation code to the assembler. The mnemonic for the operation code is "name" and must be different from all other operation codes, assembler directives and macro names. The 16-bit operation code is "value". The "type" of instruction being defined is one of the following.

TYPE	OPERANDS	LIKE
0	source,destination	MOV
2	destination	ABS
4	reg,immediate-data	LI
6	label	JMP
8	source,register	MPY
10	source,#-of-bits	LDCR
12	reg,count	SLA
14	(none)	RTWP
16	immediate-value	LWPI
18	register	STST
20	bit-displacement	SBO

Examples: define the 9995 extra instructions.

```
$OPCODE DIVS,>0180,2    DIVIDE SIGNED
$OPCODE MPYS,>01C0,2    MULTIPLY SIGNED
$OPCODE  LST,>0080,18   LOAD STATUS
$OPCODE  LWP,>0090,18   LOAD WORKSPACE POINTER
```

$REM - Macro Reminder

```
$REM                [comment]
```

This directive provides comments within a macro definition.
Examples:

```
$REM  &P1 IS LENGTH
$REM  LENGTH MUST BE LESS THAN 20
```

$SET - Set Macro Symbol

```
$SET   symbol,value  [comment]
```

This directive is used to set the value of local and/or global macro symbols. (The values of parameter and system macro symbols are set by the assembler.) The first operand of the $SET directive names the macro symbol whose value is being set. The second operand is the expression which defines the value the macro symbol is to have.

The expression is scanned and any macro symbols are replaced by their values before the expression is evaluated. The expression may be a quoted string or a numeric expression. If the expression is a numeric expression, it is evaluated then converted to a string of length 5, with leading zeros. NOTE: all arithmetic in the Assembler is done to 16 bits, that is, numbers range from 0 to 65536 with no negative numbers. In most statements, this is not a problem but it must be kept in mind when coding $SET and $IF macro directives.
Examples:

```
$SET &L1,'XYZ'           &L1='XYZ'
$SET &L2,2               &L2='00002'
$SET &G3,&L2+1           &G3='00003'
$SET &G4,'&L1ABC'        &G4='XYZABC'
$SET &G5,'&L1&L2'        &G5='XYZ00002'
$SET &L6,'&L1(2.1)'      &L6='Y'
$SET &L7,'&L1.(2.1)'     &L7='XYZ(2.1)'
$SET &G8,'&L2+1'         &G8='00002+1'
```