IBM

Series/1

SC34-0124-0

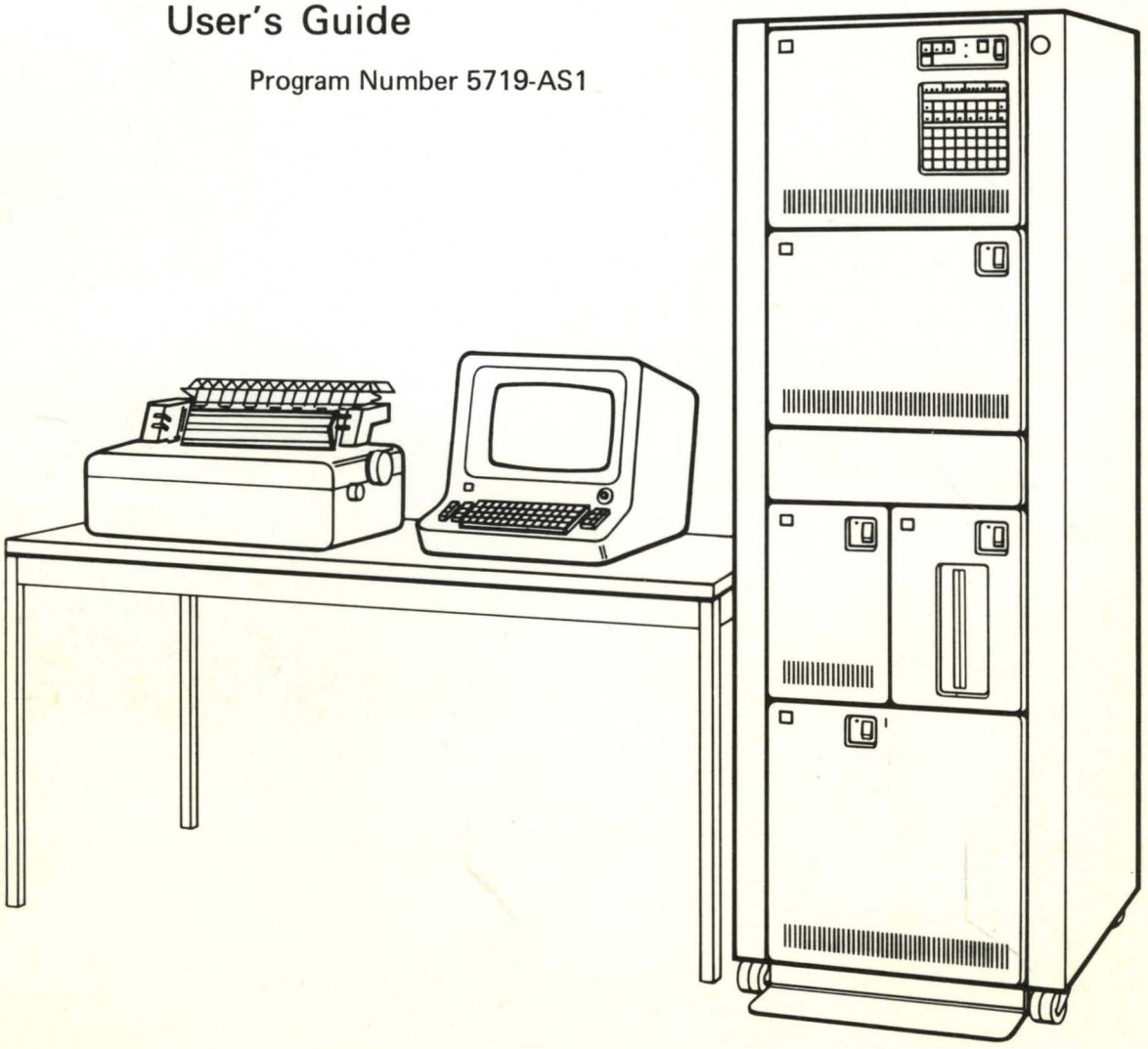
S1-21

PROGRAM
PRODUCT

ASSEMBLER USER'S GUIDE

IBM Series/1
Program Preparation Subsystem
Macro Assembler
User's Guide

Program Number 5719-AS1



IBM

Series/1

SC34-0124-0

S1-21

PROGRAM
PRODUCT

ASSEMBLER USER'S GUIDE

IBM Series/1
Program Preparation Subsystem
Macro Assembler
User's Guide

Program Number 5719-AS1

This publication is for planning purposes only. The information herein is subject to change before the products described become available.

First Edition (February 1977)

This manual applys to the IBM Series/1 Program Preparation Subsystem, program number 5719-AS1.

Significant changes or additions to the contents of this publication will be reported in subsequent revisions or Technical Newsletters. Requests for copies of IBM publications should be made to your IBM representative or the IBM branch office serving your locality.

A form for readers' comments is provided at the back of this publication. If the form has been removed, send your comments to IBM Corporation, Systems Publications, Department 27T, P.O. Box 1328, Boca Raton, Florida 33432. Comments become the property of IBM.

© Copyright International Business Machines Corporation 1977

Contents

What This Manual Can Do For You

This publication is a reference for programmers who use the IBM Series/1 assembler language. It gives specific information about assembler language functions and coding specifications.

How This Manual Is Organized

- Chapter 1 gives a brief introduction to the assembler and its features.
- Chapter 2 discusses the structure of the assembler language. It also explains the coding rules you must follow in coding an assembler-language program.
- Chapter 3 describes the characteristics of the IBM Series/1 processor. It explains register usage, addressing modes, and other information you should understand to effectively use the assembler.
- Chapter 4 describes the machine instructions. It explains the function of each instruction and how to code it. For most instructions, this chapter gives examples to help you better understand how the instructions work.
- Chapter 5 describes the assembler instructions. It explains what they do and how to code them, then gives examples of their use.
- Chapter 6 describes the macro language. Programming in macro language simplifies coding, reduces the chance for making errors, and ensures that standard sequences of instructions are coded.
- Chapter 7 describes assembler options, the program listing produced by the assembler and the control statements necessary to run and assembly. It also includes performance, invoking of the assembler, and object module formats.
- The appendixes cover structured macros, conversion tables, a priority list for assembler instructions, a summary of constants, and a summary of the macro language.

Each chapter of this publication is a separate module. This organization allows you to use the chapters as published or to combine them with information from other sources.

Each chapter has a detailed table of contents. A master index is included at the end of the manual.

What You Should Know Before You Begin

You should be familiar with the concepts of modular programming, and you should be experienced in assembler-language coding.

Related Publications

The following publications may be helpful to you. The manuals not available at this time do not show an order number and are noted by an asterisk.

IBM Series/1 Program Preparation Subsystem: Introduction, GC34-0121

IBM Series/1 Program Preparation Subsystem: Batch User's Guide *

IBM Series/1 Program Preparation Subsystem: Text Editor User's Guide *

IBM Series/1 Program Preparation Subsystem: Application Builder User's Guide *

IBM Series/1 Model 3: 4953 Processor and Processor Features Description, GA34-0022

IBM Series/1 Model 5: 4955 Processor and Processor Features Description, GA34-0021

IBM Series/1 Realtime Programming System: Macro User's Guide—Supervisor *

IBM Series/1 Realtime Programming System: Macro User's Guide—Data Management *

* These books are not presently available

Chapter 1. Introduction

Section Contents

The Assembler Language

Assembler language is a symbolic programming language that resembles machine language in form and content. It is made up of statements that represent instructions and comments. The instruction statements are the working part of the language and are divided into three groups:

- Machine instructions
- Assembler instructions
- Macro instructions

Machine Instructions

A machine instruction is the symbolic representation of a hardware instruction in the Series/1 instruction set. Machine instructions are described in Chapter 4 of this manual.

Assembler Instructions

An assembler instruction is a request to the assembler program to perform certain operations during the assembly of a source module; for example, defining data constants, defining the end of the source module, and reserving storage areas. Except for the instructions that define constants, parameter lists, or provide boundary alignment, the assembler does not translate assembler instructions into object code. The assembler instructions are described in Chapter 5 of this manual.

Macro Instructions

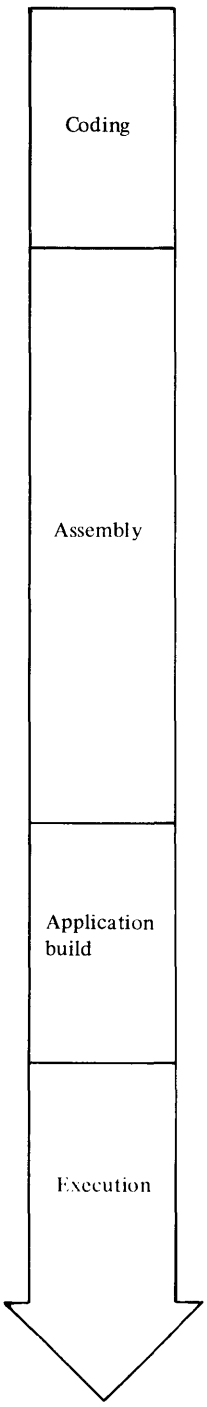
A macro instruction is a request to the assembler program to process a predefined sequence of code called a *macro definition.* From this definition, the assembler generates machine and assembler instructions which it then processes as if they were part of the original input in the source module.

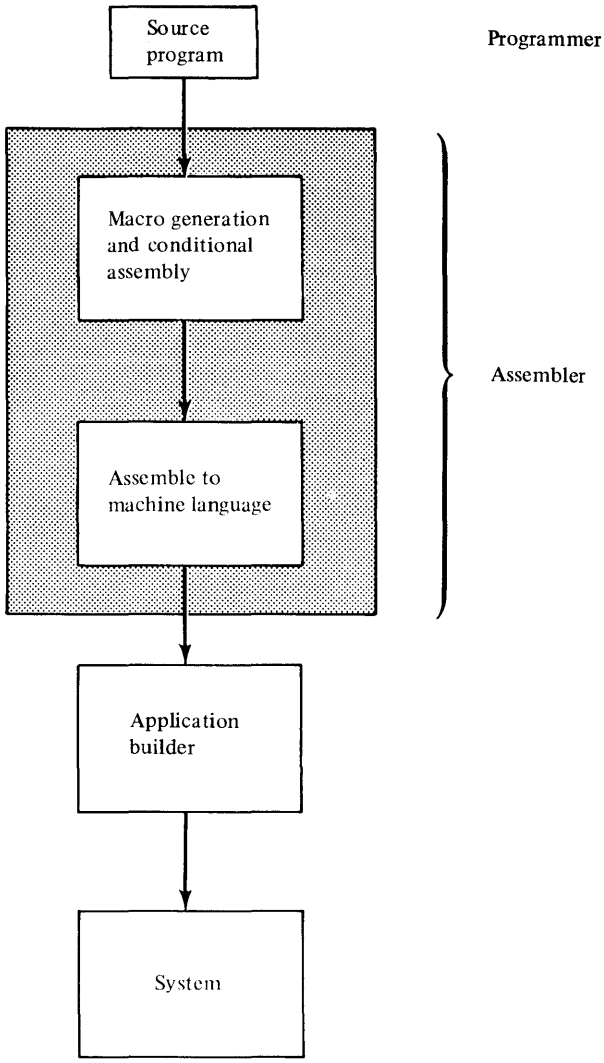
You can prepare macro definitions, then call them by coding the corresponding macro instructions. A complete description of the macro language, including the macro definition, the macro instruction, and the conditional assembly language is given in Chapter 6 of this manual.

The Assembler Program

The assembler program, also referred to as the *assembler*, processes the machine, assembler, and macro instructions you have coded in the assembler language and produces an object module in machine language.

The assembler processes the three types of assembler language instructions at different times during its processing sequence. You should be aware of this processing sequence in order to code your program correctly. The following diagram relates the assembler processing sequence to other times at which your program is processed and executed.





The assembler processes instructions at two distinct times. It processes macro instructions at preassembly time, and it processes machine instructions and assembler instructions at assembly time.

The assembler also produces information for other programs. The application builder uses such information to combine object modules and task sets which can be executed.

Coding Aids

It is normally difficult to write an assembler language program using only machine instructions. The assembler provides some coding aids that make this task easier. They are summarized next.

Symbolic Representation of Program Elements

Symbols greatly reduce programming effort and errors. You can define symbols to represent storage addresses, displacements, constants, registers, and other elements that make up the assembler language. These elements include operands, operand subfields, terms, and expressions. Symbols are easier to remember and code than numbers; also, they are listed in a symbolic cross-reference table which is printed in the program listing. Thus, you can easily find a symbol when searching for an error in your code.

Variety of Data Representation

You can use decimal, binary, hexadecimal, ASCII or EBCDIC character representation, which the assembler converts for you into the binary equivalents required by the machine instructions.

Relocatability

The assembler produces an object module that can be relocated from the originally assigned storage area to any other suitable main storage area without affecting program execution. The application builder does the relocation.

Addresses and Addressing

Each byte in main storage is specified by a unique numeric address. When you write a program, you are essentially telling the computer what addresses are involved in the operation you want it to perform. You are usually not concerned with the specific main storage locations, because the assembler keeps track of the location of the statement in your program (relative to the beginning of your program). The assembler assigns the proper address to each assembled machine-language instruction.

To keep track of the locations, the assembler uses a location counter. The counter is set to zero at the beginning of an assembly unless your program specifies otherwise. The counter is then increased by the number of bytes of main storage that each instruction needs. In this way, each statement is assigned an address relative to the beginning of the program. The assembler can then assign the addresses and displacements that are required when it produces the object program. (A displacement is the difference between the counter value of one statement and that of another.)

When your program is processed by the application builder and loaded, each statement for which storage is allocated takes on a relocated address, which is equal to the beginning main-storage address of the program *plus* the location counter value for that instruction (based on a beginning location counter value of zero).

To locate data, most machine instructions refer to a storage address. The Series/1 uses a variety of methods (called *addressing modes*) to find the data you request in your machine instructions. Addressing modes are described in Chapter 3; how you use them is discussed in Chapter 4.

Register Usage

Any one of the eight general-purpose registers can be used to hold a value, an address, or a displacement for manipulating data, maintaining counters, or determining the address of a particular instruction or storage location.

The instruction address register contains the address of the next instruction to

be executed unless a branch or jump instruction breaks the normal sequence. When this occurs, the contents of the instruction address register change because the program transfers control to an instruction not immediately following the current instruction.

Segmenting a Program

You can code a program in sections and later combine the sections into the executable program. Sections are assembled in any combination, individually or grouped. You arrange the sections in the order required for proper execution of the program during the combining process. The combining process is called *linking* and is performed by the application builder program.

Dividing a large program into several sections and assemblies has certain advantages:

- More than one programmer can code sections of the program. Each can assemble and debug his sections independently of the others.
- The linking process is much faster, in terms of computer time, than the assembly process. You can assemble a section of the program and link the new section to the already assembled program. This uses less computer time than assembling the entire program.
- Sections that are common to more than one program are assembled only once. You can then link the common sections to the unique sections of each program. You again reduce computer time and also have shorter assembly listings that are easier to debug.
- You can configure a program to various main storage requirements much more easily by linking the sections into different combinations of storage loads or phases. Of course, you must make provisions for these variations in your program logic.

Four types of program sections can be defined in the assembler language—control sections, common sections, global sections, and dummy sections. Control sections define the object code, that is, machine instructions and data definitions. A common section defines an area of main storage that can be shared with the program sections in multiple assemblies within a task. A global section may define either task set or system global area. Task set global is addressable by all programs executing in a partition. If a task set issues a request to transfer control to another task set, the portion of task set global common to both task sets is not overlayed. System global is contained in the shared task set and is addressable by all programs link-edited against the shared task set. The application builder determines whether a global section is a task set or system global, according to the name associated with the global section. If the named global exists in the shared task set and has the proper attributes (for example, length), the shared task set global area will be used, otherwise, a global area will be generated within the task set. For more information on task sets, reference the *Application Builder User's Guide*. A dummy section describes to the assembler the format of data located elsewhere. Dummy sections do not appear in the output module of an assembly. The linking process combines only control, common and global sections.

The following assembler instructions define the beginning and end of the various sections in your assembly.

- START and CSECT instructions define the beginning of a control section. START is used only to define the first control section in an assembly.
- The COM instruction defines the beginning of a common section.
- The GLOBL instruction defines the beginning of a global section.
- The DSECT instruction defines the beginning of a dummy section.
- The end of any type of section is defined when another section is started. The END instruction defines the end of all sections in an assembly.

Linkage Between Source Modules

You can create symbolic linkages between separately assembled source modules. This allows you to refer symbolically from one source module to data defined in another source module. You can also use symbolic addresses to branch between modules.

Combining separately assembled control sections successfully depends on the techniques for symbolic linkages between the control sections. For example, symbols can be defined in one module and referred to in another. The application builder then completes the linkage, using the information passed to it by the assembler. Not only is the linkage symbol defined (used as a name), it must also be identified to the assembler by means of an ENTRY assembler instruction unless the symbol is the name of a CSECT or START statement. After a symbol is identified as one that names an entry point, another module can then use that symbol to bring about a branch operation or a data reference.

A module that refers to a linkage symbol defined in another module must identify it as an externally defined symbol used to bring about linkage. The EXTRN or WXTRN assembler instructions identify such symbols.

Symbolic linkage can also be achieved by means of the V-type or W-type address constant, or by means of the BALX or BX machine instruction. Each constant is an external reference since it is created from an externally defined symbol that need not be identified by an EXTRN or WXTRN statement. The V-type or W-type address constant can be used to branch to other modules or to refer to data in those modules.

Program Listing

The assembler produces a listing of your source module, including any generated statements, and the object code assembled from the source module. You can control the content of the listing to a certain extent. The assembler also prints messages about actual errors and warnings about potential errors in your source module. A complete description of the program listing is given in Chapter 7 of this manual.

Programmer Procedures

This book gives you information on the assembler language and on the execution of the assembler. Using Series/1 involves much more. This section gives you an outline of the process required to generate and execute a program. A step by step procedure, which refers to the flowchart on page 1-7, points you to the library source containing more information.

Step 1. Design Application and Support System

You must combine your knowledge of designing a realtime system with the Series/1 to produce a specification furnishing information for all succeeding steps.

An introduction to the Series/1 programming support and references to detailed information is in the *IBM Series/1 Realtime Programming System: Introduction and Planning Guide*, GC34-0102.

Step 2. Generate Realtime Operating System

You can generate your specific operating system by using information in the *IBM Series/1 Realtime Programming System: Generation and Installation Procedures* *.

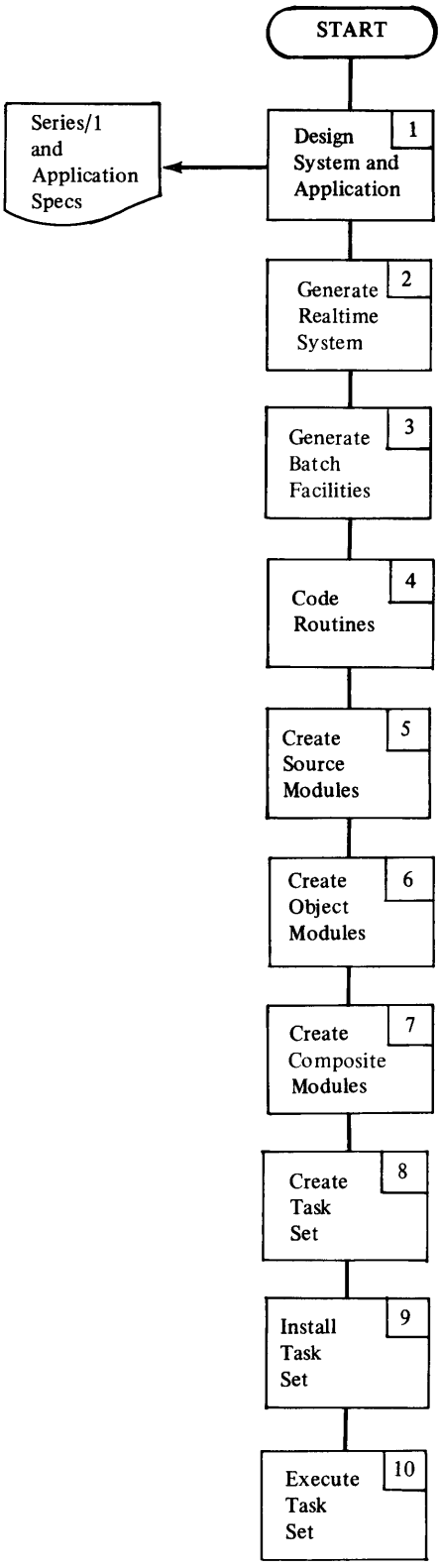
Step 3. Generate Batch Processing and Program Preparation Facility

You can generate your specific system by using information in the *IBM Series/1 Program Preparation Subsystem: Batch User's Guide* *.

Step 4. Code Assembler Language Programs

This book contains information on the coding of all assembler language statements and on structured macros furnished as a part of the assembler. You may also need to code macro statements for macros defined specifically for your application and code system support macros furnished as a part of that program product.

Information on these macros is in three volumes: *IBM Series/1 Realtime Programming System: Macro User's Guide—Supervisor* *; *IBM Series/1 Realtime Programming System: Macro User's Guide—Data Management* *; *IBM Series/1 Realtime Programming System: Macro User's Guide—Communications* *.


START
Series/1 and Application Specs
Design System and Application
1
Generate Realtime System
2
Generate Batch Facilities
3
Code Routines
4
Create Source Modules
5
Create Object Modules
6
Create Composite Modules
7
Create Task Set
8
Install Task Set
9
Execute Task Set
10

Step 5. Create Source Modules

State any special requirements for source modules in your application specification. Information on normal requirements for source modules assembled by the Series/1 macro assembler is in this book and in the *IBM Series/1 Program Preparation Subsystem: Text Editor User's Guide* *.

Step 6. Create Object Modules

State any special requirements for object modules in your application specification. Information on normal requirements for object modules processed by the Series/1 application builder is in this book.

Step 7. Create Composite Modules

State any special requirements for composite modules in your application specification. Requirements for composite modules which are used in building task sets to be executed under Series/1 realtime operating system are in the *IBM Series/1 Program Preparation Subsystem: Application Builder User's Guide* *.

Step 8. Create Task Sets

State the requirements for task sets in your application specification. Information on using the application builder to meet these requirements is in the *IBM Series/1 Program Preparation Subsystem: Application Builder User's Guide* *.

Step 9. Install Task Sets

State any requirements for installation of task sets in your application specification. Information on how to meet these requirements is in the *IBM Series/1 Realtime Programming System: Macro User's Guide—Supervisor* *.

Step 10. Execute Task Sets

State requirements for executing your application task sets in your application specification. Information to help you meet these requirements is in the *IBM Series/1 Realtime Programming System: Operator Commands and Utilities* * and *IBM Series/1 Program Preparation Subsystem: Batch User's Guide* *

Step 11. Debug Task Sets

To debug task sets reference one of the following manuals depending on your specific problem. The *IBM Series/1 Realtime Programming System: Control Blocks and Debugging Guide* *, gives you error log descriptions, and explains how to get a storage dump. Information on macro errors is found in the *IBM Series/1 Realtime Programming System Macro Reference* *. For descriptions of control blocks reference the Program Logic Manual (PLM) for your particular function. Explanations for messages and codes are found in the *IBM Series/1 Program Preparation Subsystem: Messages and Codes* * manual and/or *IBM Series/1 Realtime Programming System: Messages and Codes* *.

* These books are not presently available.

Summary of Programmer Procedures

The table below gives a summary of the procedures to tie them to processes or publications in Series/1. For a detailed overview of Series/1 and a comparison of how system functions are supported by different languages, refer to the *IBM Series/1 Realtime Programming System: Introduction and Planning Guide*, SC34-0102.

Programmer Procedure	*Associated System Step*	*Publication*
1. Planning and System Design	None.	*IBM Series/1 Realtime Programming System: Introduction and Planning Guide*, GC34-0102
2. Building the Realtime Building the Realtime Operating System	System Generation	*IBM Series/1 Realtime Programming System: Generation and Installation Procedures* *
3. Building the Program Preparation & Batch Processing System	System Generation	*IBM Series/1 Program Preparation Subsystem: Batch User's Guide* *
4. Programs	None. (Coding sheet)	*IBM Series/1 Program Preparation Subsystem: Macro Assembler User's Guide*, SC34-0124 *IBM Series/1 Realtime Programming System: Macro User's Guide—Supervisor* * *IBM Series/1 Realtime Programming System: Macro User's Guide—Data Management* * *IBM Series/1 Realtime Programming System: Macro User's Guide—Communications* * *IBM Series/1 Realtime Programming System: Macro Reference* *
5. Creating Source Modules (On diskette)	Text Editor (Note. Or, use 3741)	*IBM Series/1 Program Preparation Subsystem: Text Editor's User's Guide* *
6. Creating Object Modules	Assembler	*IBM Series/1 Program Preparation Subsystem: Macro Assembler User's Guide*, SC34-0124
7. Creating Composite Modules	Application Build	*IBM Series/1 Program Preparation Subsystem: Application Builder User's Guide* *
8. Building Task Sets	Application Build: Phases 1 & 2 & 3	*IBM Series/1 Program Preparation Subsystem: Application Builder User's Guide* *
9. Installing Task Sets (Not required)	A series of steps using the utility DEFINE and COPY facilities and the install option on the start taskset operator command	*IBM Series/1 Realtime Programming System: Macro User's Guide—Supervisor* *

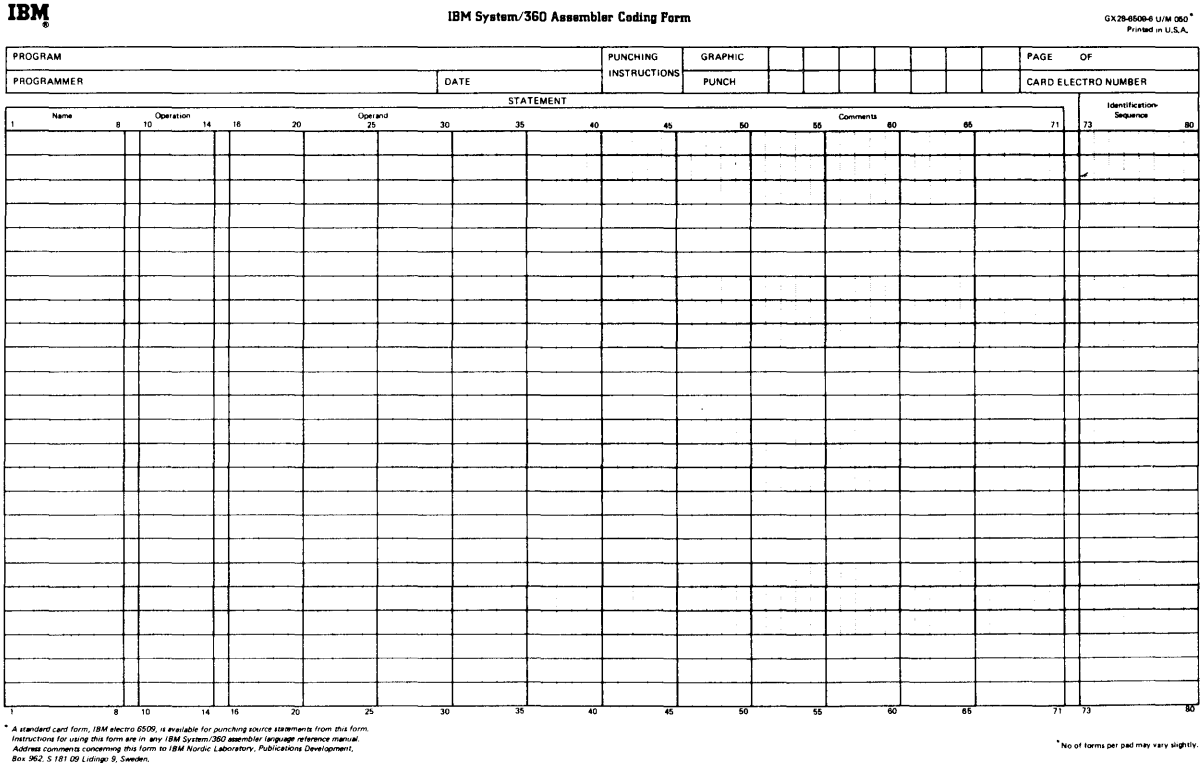
10. Executing Task Sets	Direct: STARTTASK operator command or supervisor macro	*IBM Series/1 Realtime Programming System: Operator Commands and Utilities* * *IBM Series/1 Realtime Programming System: Macro User's Guide—Supervisor* *
	Batch: Job Stream Processor	*IBM Series/1 Program Preparation Subsystem: Batch User's Guide* *
11. Debugging Task Sets	Error Management	*IBM Series/1 Realtime Programming System: Control Blocks and Debugging Guide* *
	Messages and Return Codes	*IBM Series/1 Program Preparation Subsystem: Messages and Codes* * and *IBM Series/1 Realtime Programming System: Messages and Codes* *
*These books are not presently available.		

Chapter 2. Coding and Structure of the Assembler Language

Section Contents

Coding Conventions

This chapter describes the coding conventions that you must follow in writing assembler language programs. Assembler language statements are usually written on a coding form before they are entered as source statements through some form of input to the system. You can write assembler language statements on the standard coding form (GX28-6509). Columns 1–72 of this form correspond to the positions on a source statement entered at the operator station.


IBM

IBM System/360 Assembler Coding Form

GX28-6509-6 U/M 050* Printed in U.S.A.

PROGRAM | PUNCHING INSTRUCTIONS | GRAPHIC | PAGE OF

PROGRAMMER | DATE | PUNCH | CARD ELECTRO NUMBER

STATEMENT

Name | Operation | Operand | Comments | Identification-Sequence

1 8 10 14 16 20 25 30 35 40 45 50 55 60 65 71 73 80

* A standard card form, IBM electro 6509, is available for punching source statements from this form. Instructions for using this form are in any IBM System/360 assembler language reference manual. Address comments concerning this form to IBM Nordic Laboratory, Publications Development, Box 962, S 181 09 Lidingö 9, Sweden.

* No of forms per pad may vary slightly.

Field Boundaries

Assembler language statements usually occupy one 80-column line on the standard form. (For statements occupying more than 80 columns, see "Continuation Lines" in this chapter.) Note that any printable character entered at a position in a source statement is reproduced in the listing printed by the assembler. Each line of the coding form is divided into three main fields:

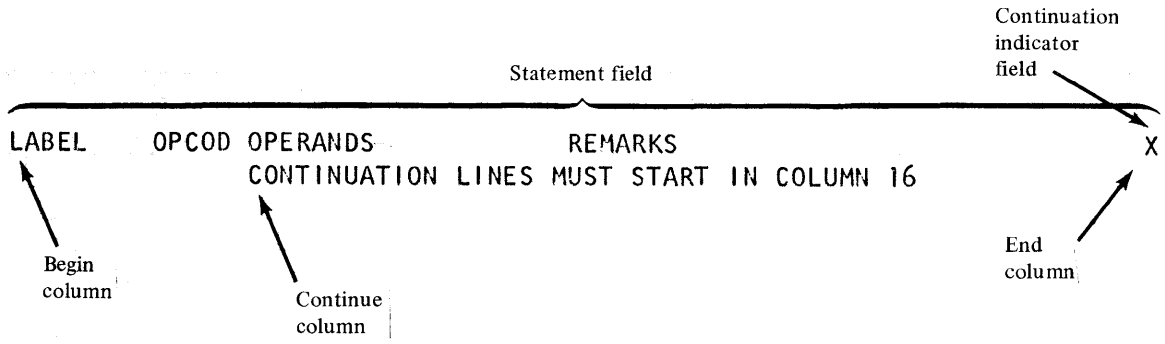
- The statement field
- The continuation indicator field.
- The identification and sequence field

The Statement Field

The instructions and comments statements must be written in the statement field. The statement field starts in the "begin" column and ends in the "end" column. Any continuation lines needed must start in the "continue" column and may extend to the "end" column. The assembler assumes the following standard values for these columns:

- The "begin" column is column 1
- The "end" column is column 71
- The "continue" column is column 16.

These standard values can be changed by using the ICTL instruction. However, all references to the "begin", "end", and "continue" columns in this manual refer to the standard values previously described.



The Continuation Indicator Field

The continuation indicator field occupies the column following the end column. Therefore, the standard position for this field is column 72. A nonblank character in this column indicates that the current statement is continued on the next line. This column must be blank if a statement is completed on the same line; otherwise the assembler will treat the statement that follows on the next line as a continuation line of the current statement.

The Identification and Sequence Field

The identification and sequence field can contain identification characters, or sequence numbers, or both. If you have coded the ISEQ instruction, the assembler will verify whether or not the source statements are in the correct sequence.

Note. The ISEQ instruction is used to define the identification and sequence field column boundaries (there are no 'standard' assembler default boundaries for this field). However, column 73 through 80 are set aside for this purpose on the standard coding form, and so are normally used. The single requirement is that this field must not be specified to be within the statement field boundaries.

Field Positions

The statement field is always coded between the begin and the end columns. The continuation indicator field is always coded in the column after the end column. The identification and sequence field usually is the field after the continuation indicator field. However, the ICTL instruction, by changing the standard begin, end, and continue columns can create a field before the begin column. This field can then contain the identification and sequence field.

Continuation Lines

To continue a statement on another line:

1. Enter a nonblank character in the continuation indicator field (column 72). This character is not treated as part of the statement coding. An operand field can be continued by coding it up through column 71, or by terminating it with a comma followed by at least one blank.
2. Continue the statement on the next line, starting in the continue column (column 16). The columns prior to the continue column (columns 1–15) must be blank.

Only one continuation line is allowed for a single assembler language statement. However, macro instruction statements and the prototype statement of macro definitions can have as many continuation lines as needed. When more than one continuation line is needed, enter a nonblank character in the continuation indicator field of each line that is to be continued.

Comments Statement Format

Comments statements are not assembled as part of the object module, but are only printed on the assembly listing. You can write as many comments statements as you need, as long as you follow these rules:

- Comments statements require an asterisk in the begin column.

  Note. Internal macro definition comments statements require a period in the begin column, followed by an asterisk in the next column (for details see "Internal Macro Comments Statements" in this chapter).

- You can use any characters, including blanks and special characters, of the character set.
- Comments statements cannot be continued. Code comments statements in the statement field and do not let them run over into the continuation indicator field.
- Comments statements must not appear between an instruction statement and its continuation lines.

Instruction Statement Format

The statement field of an instruction statement must include an operation entry and can contain one or more of the following entries:

- A name entry
- An operand entry
- A remarks entry

The standard coding form is divided into fields that provide fixed positions for the name, operation, and operand entries.

1. An 8-character name field starting in column 1
2. A 5-character operation field starting in column 10
3. An operand field that begins in column 16

```
LABEL    MVW   R3,ADCON          REMARKS ENTRY
         BAL   ADDRESS,R7        NAME ENTRY OMITTED
SECTDEF  CSECT                   OPERAND ENTRY NOT REQUIRED
         ORG   ,                 OPERAND ENTRY OMITTED
```

Adherence to these field positions is called *fixed format.*

It is not necessary to code the operation and operand entries according to the fixed fields on the standard coding form. Instead, you can write these entries in any position, called *free format*, subject to certain formatting specifications.

Whether you use fixed or free format, the following general rules apply when you code an instruction statement:

- Write the entries in the following order: name, operation, operand, and remarks.
- The entries must be contained in the begin column (1) through the end column (71) of the first line and, if needed, in the continue column (16) through the end column (71) of any continuation lines.
- The entries must be separated from each other by one or more blanks.
- If used, the name entry must start in the begin column (normally column 1).
- The name and operation entries, each followed by at least one blank, must be in the first line of an instruction statement.
- The operation entry must start to the right of the begin column.

```
LABEL    MVW   R3,ADCON            FIXED FORMAT STATEMENT.

LABEL MVW R3,ADCON FREE FORMAT STATEMENT.

LABEL    MVW                                                    X
               R3,ADCON ONLY OPERANDS AND REMARKS ALLOWED HERE.

 MVW R3,ADCON NAME ENTRY OMITTED, COLUMN 1 MUST BE BLANK.

LABEL    MVW   R3,                 CONTINUE OPERANDS            X
               ADCON               ON NEXT LINE
```

The *name entry* identifies an instruction statement. The following rules apply to the name entry:

- It is usually optional.
- It must be a valid symbol of 1–8 characters at assembly time (after substitution for variable symbols, if specified on model statements within macro definitions).

The *operation entry* is the symbolic operation code that specifies the machine, assembler, or macro instruction to be processed. The following rules apply to the operation entry:

- It is mandatory.
- For machine and assembler instructions it must be a valid symbol at assembly time (after substitution for variable symbols, if specified on model statements within macro definitions). The standard symbolic operation codes are 5 characters or less.
- For macro instructions it can be any valid symbol of 1–8 characters that is not identical to the operation codes for machine and assembler instructions.

The *operand entry* has one or more operands that identify and describe the data used by an instruction. The following rules apply to operands:

- One or more operands are usually required, depending on the instruction.
- Operands must be separated by commas. No blanks are allowed between the operands and the commas that separate them. When an operand entry is being

continued on the next line, the last operand on the first line can be terminated with a comma followed by one or more blanks.

- Operands must not contain embedded blanks, because a blank normally indicates the end of the operand entry. However, blanks are allowed if they are included in character strings enclosed in apostrophes (for example, C' JN') or in logical expressions (see "Logical (SETB) Expressions" in Chapter 6).

You can use a *remarks entry* to comment on the current instruction. The following rules apply to the remarks entry:

- It is optional.

* These manuals not presently available.

- It can contain any of the 256 characters of the character set, including blanks and special characters.
- It can follow any operand entry.
- If an entire operand entry is omitted, remarks are allowed if the absence of the operand entry is indicated by a comma preceded and followed by one or more blanks.

Note. Macro prototype statements and macro instructions without operands cannot have a remarks entry, even if a comma is coded as described above.

Assembler Language Structure

This section describes the structure of the assembler language (the various statements allowed in the language and the elements that make up those statements).

A *source module* is a sequence of instruction and comment statements that make up the input to the assembler. There are 3 types of instruction statements:

- *Machine instructions*—symbolic representation of machine language instructions, which the assembler translates into machine language code
- *Assembler instructions*—instructions to the assembler program to perform certain operations during assembly of a source module, such as defining data constants and reserving storage areas
- *Macro instructions*—instructions to the assembler program to process predefined sequences of code called *macro definitions* (from which the assembler generates machine and assembler instructions which it then processes as if they were part of the original input source module)

The operand field of machine instructions is composed of *expressions*, which are composed of *terms* and combinations of terms. Remarks on the instruction statements and comments statements are composed of *character strings*. Terms and character strings are both composed of characters. The following paragraphs define these language elements.

Character Set

Terms, expressions, and character strings used to build source statements are written with the following characters:

- Alphameric characters
  - Alphabetic characters A through Z
  - Special characters $, #, and @
  - Digits 0 through 9
- Special characters
  + − , = . * ( ) ' / & blank

Note. Character strings can contain any of the 256 characters of the character set.

Normally, strings of alphameric characters are used to represent data, and special characters are used as:

- Arithmetic operators in expressions
- Data or field delimiters
- Indicators to the assembler for specific handling

Terms

A term is the smallest element of the assembler language that represents a distinct and separate value. It can therefore be used alone or in combination with other terms to form expressions. Terms have absolute or relocatable values that are assigned by the assembler or are inherent in the terms themselves.

A term is *absolute* if its value does not change upon program relocation and is *relocatable* if its value can be modified to compensate for a change in program origin.

The types of terms are:

- *Symbols*—absolute or relocatable; value is assigned by the assembler
- *Location counter reference*—relocatable; value is assigned by the assembler
- *Symbolic parameter attributes*—absolute; value is assigned by the assembler
- *Self-defining terms*—absolute; value is inherent in term

Symbols

You can use a symbol to represent storage locations or arbitrary values. You can write a symbol in the name field of an instruction and and then specify this symbol in the operands of other instructions, thus referring to the former instruction symbolically. This symbol represents a relocatable address.

You can also assign an absolute value to a symbol by coding it in the name field of an EQU or EQUR instruction with an operand whose value is absolute. Symbols in the name field of EQUR instructions can be used in other instruction operands to represent registers; symbols in the name field of EQU instructions can be used in other instruction operands as displacements in explicit addresses, immediate data, lengths, and implicit addresses with absolute values. The advantages of symbolic numeric representation are:

- You can remember symbols more easily than numeric values, thus reducing programming errors and increasing programming efficiency.
- You can use meaningful symbols to describe the program elements they represent; for example, INPUT can name a field that is to contain input data, or INDEX can name a register to be used for indexing.
- You can change the value of one symbol (through an EQU instruction) more easily than you can change several numeric values in many instructions.
- Symbols are entered into a cross-reference table that the assembler optionally prints in the program listing.

The symbol cross-reference table helps you to find a symbol in a program listing, because it lists (1) the number of the statement in which the symbol is defined (that is, used as the name entry), and (2) the numbers of all the statements in which the symbol is used in the operands.

The Symbol Table

The assembler maintains an internal table called a *symbol table*. When the assembler processes your source statements for the first time, the assembler assigns an absolute or relocatable value to every symbol that appears in the name field of an instruction. The assembler enters this value, which normally reflects the setting of the location counter, into the symbol table; it also enters the attributes with the data represented by the symbol. The values of the symbol and

its attributes are available later when the assembler finds this symbol used as a term in an operand or expression.

The assembler recognizes three types of symbols:

- *Ordinary symbols*—used in the name and operand fields of machine and assembler instruction statements; written as an alphabetic character followed by 0–7 alphameric characters (no blanks allowed). For example: BEGIN
- *Variable symbols*—used only in macro processing conditional assembly instructions; written as an ampersand followed by an alphabetic character followed by 0–6 alphameric characters (no blanks allowed). For example: &PARAM
- *Sequence symbols*—used only in macro processing conditional assembly instructions; written as a period followed by an alphabetic character followed by 0–6 alphameric characters. For example: .SEQ01

An ordinary symbol is considered defined when it appears as:

- The name entry in a machine or assembler instruction of the assembler language, or
- One of the operands of the following instructions: EXTRN, WXTRN.

Ordinary symbols that appear in instructions generated from model statements at preassembly time are also considered defined.

The assembler assigns a value to the ordinary symbol in the name field as follows.

- According to the address of the leftmost byte of the storage area that contains one of the following:
  - Any machine instruction
  - A storage area defined by the DS instruction
  - Any constant defined by the DC instruction

  The address value thus assigned is relocatable, because the object code assembled from these items is relocatable.
- According to the value of the expression specified in the operand of an EQU instruction. This expression can have a relocatable or absolute value, which is then assigned to the ordinary symbol.
- According to the value of the expression specified in the operand of an EQUR instruction. This expression must have an absolute value in the range 0–7, which is then assigned to the register symbol.

The value of an ordinary symbol must be representable in 16 bits.

Note. The symbol table can contain a maximum of 57330 entires. In a single assembly, the total number of ordinary symbol entries plus ESD entries cannot exceed this maximum. ESD entries are created for:

- control sections
- dummy sections
- global sections
- common sections
- unique symbols in EXTRN, WXTRN, and ENTRY statements
- V, W, and N-type address constants
- BALX and BX instructions.

Restrictions on Symbols

Predefined Register Symbols

The following symbols are predefined by the assembler and reserved for use only as register symbols:

- R0 (general-purpose register 0)
- R1 (general-purpose register 1)
- R2 (general-purpose register 2)

- R3 (general-purpose register 3)
- R4 (general-purpose register 4)
- R5 (general-purpose register 5)
- R6 (general-purpose register 6)
- R7 (general-purpose register 7)
- FR0 (floating-point register 0)
- FR1 (floating-point register 1)
- FR2 (floating-point register 2)
- FR3 (floating-point register 3)

These symbols are absolute and used only for register reference in machine and assembler instruction operands. Any other usage causes an error message to be generated. Predefined register symbols appear in the cross-reference listing.

Unique Definition

A symbol must be defined only once in a source module (even a source module that contains two or more control sections) with the following exceptions:

- You can use a duplicate symbol as the name entry of a CSECT, GLOBL, COM or DSECT instruction. The first use identifies the beginning of the control section, and subsequent uses identify continuations of the control section. A symbol used in the name field of one type of section may not be repeated on another type. For example, a DSECT and a GLOBL statement may not have the same name field. A label appearing on a START statement may be used on a subsequent CSECT statement.
- A symbol can appear more than once in the operands of the following instructions:
  - ENTRY
  - BALX
  - BX
  - EXTRN
  - WXTRN
  - DC for V-, W-, or N-type address constants provided the attributes are not contradictory (that is, the same symbol can be repeated in an EXTRN and BALX instruction but cannot be repeated in a WXTRN and EXTRN).

Note. An ordinary symbol that appears in the name field of a TITLE instruction is not a definition of that symbol. It can, therefore, be used in the name field of any other statement in a source module.

Previously Defined Symbols

The symbols used in the operands of the following instructions must be defined in a previous instruction:

- EQU
- EQUR
- ORG
- DC and DS (in modifier and duplication factor expressions)

The following sample code indicates the ways symbols can be defined and used:

```
FIRST    START 128                FIRST CONTROL SECTION STARTS HERE.
         EXTRN READER,PRINTER     SYMBOLS DEFINED IN EXTRN STATEMENT.
         :
XR3      EQUR  3
INDEX    EQUR  XR3                'XR3' IS PREVIOUSLY DEFINED.
XR4      EQUR  4
         :
ENTRIES  MVW   TABLE,XR3          SYMBOL USED IN NAME AND OPERAND.
         :
TABLE    DS    F                  SYMBOL IN NAME FIELD OF DS.
         :
SECOND   CSECT                    SECOND CONTROL SECTION STARTS HERE.
         :
         MVW   XR4,ADCON          SYMBOL IN OPERAND FIELD.
         :
ADCON    DC    A(READER)          SYMBOL IN NAME FIELD OF DC.
         :
FIRST    CSECT                    RESUME FIRST CONTROL SECTION.
         :
         END
```

The unique symbols, in the order they were defined, are:

FIRST
READER
PRINTER
XR3
INDEX
XR4
ENTRIES
TABLE
SECOND
ADCON

Location Counter Reference

The assembler maintains a location counter to assign storage addresses to your program statements. You can refer to the current value of the location counter at any place in a source module by specifying an asterisk as a term in an operand.

As the instructions and constants of a source module are being assembled, the location counter has a value that indicates a location in storage. The assembler increases the location counter according to the following rules:

- As each instruction or constant is assembled, the location counter increases by the length of the assembled item.

- The location counter always points to the first byte of the instruction being assembled.
- All references to the location counter in the operand field are relative to the first byte of the instruction being assembled.
- If the statement is named by a symbol, the value of the symbol is the value of the location counter.

The assembler maintains a location counter for each control section in the source module. (For complete details about the location counter setting in control sections, see "Program Sectioning" in Chapter 5.) The assembler maintains the internal location counter as a 16-bit value. If you specify addresses greater than 65,535, the assembler issues the error message 'CPA205S LOCATION COUNTER ERROR'.

You refer to the location counter reference by coding an asterisk (*). Code an asterisk as a relocatable term only in the operands of:

- Machine instructions
- DC and DS instructions
- EQU, ORG, and USING instructions

The value of the location counter reference (*) is the current value of the location counter when the asterisk is specified as a term. The asterisk has the same value as the address of the first byte of the instruction being assembled. For the value of the asterisk in address constants with duplication factors, see "A-type Address Constant" in Chapter 5.

Coding an asterisk in the operand of an assembler language instruction or a machine instruction (as part of an address) is the same as placing a symbol in the name field of the same statement and then using that symbol in the operand. Be careful how you use this technique; inserting or deleting instructions between an instruction and the location it refers to makes the displacement from the location counter invalid.

```
LOCATION EQU   *
         AWI   -1,R1
         :
         B     LOCATION          BRANCH TO AWI INSTRUCTION
         :
LOCAD    DC    A(LOCATION)       ADDRESS OF AWI INSTRUCTION
         :
LOC2     DC    A(*)              ADDRESS OF LOC2--SAME AS CODING:
*                                LOC2  DC  A(LOC2)
```

Symbol Length Attribute Reference

When you reference the length attribute of a symbol, you get the length of the instruction or data referred to by the symbol. You can use this reference as a term in instruction operands to:

- Specify storage area lengths
- Cause the assembler to compute length specifications
- Build expressions to be evaluated by the assembler.

Specifications for Length Attribute References

You must code a length attribute reference according to the following rules.

- The format must be L' immediately followed by a valid symbol.
- The symbol must be defined in the same source module in which the symbol length attribute reference is specified.
- The symbol length attribute reference can be used in the operand of any instruction that requires an absolute term.

The value of the length attribute is normally the length, in bytes, of the storage area required by an instruction, constant, or field represented by a symbol. The assembler stores the value of the length attribute in the symbol table along with the address value assigned to the symbol. When the assembler encounters a symbol length attribute reference, it substitutes the value of the attribute from the symbol table entry for the symbol specified in the reference.

The assembler assigns the length attribute values to symbols in the name field of instructions as follows:

- For machine instructions, it assigns a value of 2, 4 or 6 depending on the length of the instruction.
- For the DC and DS instructions, it assigns either the length or explicitly specified length of the first operand. The length attribute is not affected by a duplication factor.
- For the EQU instruction, it assigns the length attribute value of the leftmost or only term in the operand.

The length attribute of a self-defining term is always one. The length attribute of another length attribute is always one. See the length specifications for the individual instructions in Chapter 5. Note that the value of the length attribute is available only at assembly time.

```
                                                                    Value of symbol
                                                                    length attribute
                                                                    (at assembly time)

MACHA    MVW   DISP,R5          GENERATES A FOUR BYTE INSTRUCTION.   L'MACHA = 4
MACHB    IR    R1,R3            GENERATES A TWO BYTE INSTRUCTION.    L'MACHB = 2
         :
         :
DISPO    DC    A(OTHER)         IMPLICIT LENGTH OF TWO BYTES         L'DISPO = 2
CHAR4    DC    C'ABCD'          IMPLICIT LENGTH OF FOUR BYTES        L'CHAR4 = 4
DUPLC    DC    3F'200'          IMPLICIT LENGTH NOT AFFECTED BY DUP. L'DUPLC =2
INPUT    DS    CL40             EXPLICIT LENGTH SPECIFIED.           L'INPUT =40
AREA1    DS    OCL40            NOT AFFECTED BY DUPLICATION FACTOR.  L'AREA1 =40
AREA2    DS    40C              IMPLICIT LENGTH OF ONE BYTE          L'AREA2 = 1
AREA3    DS    4XL2             EXPLICIT LENGTH NOT AFFECTED BY DUP. L'AREA3 = 2
         :                 Leftmost or only term
         :                 <-
CABCD    EQU   CHAR4            SINGLE TERM.                         L'CABCD = 4
TERM2    EQU   CHAR4+2          TWO TERMS, LENGTH OF LEFTMOST ONLY.  L'TERM2 = 4
TERM3    EQU   AREA2-AREA1      TWO TERMS, LENGTH OF LEFTMOST ONLY.  L'TERM3 = 1
TERM4    EQU   TERM3            SINGLE TERM.                         L'TERM4 = 1
         :
         :
SDTM1    EQU   102              SINGLE SELF-DEFINING TERM.           L'SDTM1 = 1
SDTM2    EQU   X'FF'+A-B        MULTIPLE TERMS, LEFTMOST IS SELF-
*                               DEFINING.                            L'SDTM2 = 1
SDTM3    EQU   C'AB'            CHARACTER SELF-DEFINING TERM.        L'SDTM3 = 1
         :
         :
LOCAT    EQU   *+2              LOCATION COUNTER REFERENCE.          L'LOCAT = 1
LNGH1    DC    A(L'LNGH1)       LENGTH OF CURRENT INSTRUCTION.       L'LNGH1 = 2
         DC    CL2'TEXT'
LOADX    MVWI  L'LOADX,R4       LENGTH OF CURRENT INSTRUCTION.       L'LOADX = 4
LATTR    EQU   L'LOADX          LENGTH OF A LENGTH ATTRIBUTE.        L'LATTR = 1
```

Other Attribute References

Other attributes describe the characteristics and structure of the data you define in a program (for example, the kind of constant you specify or the number of characters you need to represent a value). These attributes are the type (T), count (K), and number (N) attributes.

You can refer to these attributes only in macro definition statements; for full details, see "Data Attributes" in Chapter 6.

Self-defining Terms

A self-defining term lets you specify a value explicitly. With self-defining terms, you can specify decimal, binary, hexadecimal, EBCDIC character data or ASCII character data. These terms have absolute values and can be used as absolute terms in expressions to represent bit configurations, absolute addresses, displacements, length or other modifiers, and duplication factors.

Self-defining terms:

- Represent machine language binary values
- Are absolute terms; their values do not change upon program relocation
- Are padded on the left with zeros if less than one word

The assembler maintains the values represented by self-defining terms to 16 bits; self-defining terms are always considered as positive values in the range zero through 65,535.

A *decimal self-defining term* is an unsigned decimal number. The assembler allows:

- High-order zeros
- A maximum of five decimal digits
- A range of values from zero through 65,535

Note. A negative number is specified as an expression. For details, see "Expressions" later in this chapter.

```
XR3     EQUR  ØØ3          HIGH-ORDER ZEROS.
SPACE   EQU   65535        5 DIGITS IS MAXIMUM VALUE.
```

A *binary self-defining term* must be coded as the letter B followed by 1–16 binary digits enclosed in apostrophes. For example:

Binary self-defining term	*Binary value*
B'1111100'	00000000 01111100
B'100'	00000000 00000100
B'1'	00000000 00000001

The assembler assembles each binary digit exactly as specified.

A *hexadecimal self-defining term* must be coded as the letter X followed by 1–4 hexadecimal digits enclosed in apostrophes. For example:

Hexadecimal self-defining term	*Binary value*
X'FFA0'	11111111 10100000
X'F'	00000000 00001111
X'C01'	00001100 00000001
X'7FFF'	01111111 11111111
X'8000'	10000000 00000000
X'0'	00000000 00000000

The assembler assembles each hexadecimal digit into its 4-bit binary equivalent as shown above, and allows a range of values from X'0000' through X'FFFF'.

An *EBCDIC character self-defining term* must be coded as the letter C, followed by 1 or 2 characters enclosed in apostrophes. When assembling EBCDIC character constants, the assembler:

- Allows any of the 256 8-bit combinations as input. This includes the printable characters, including blanks and special characters.
- Assembles each character into its 8-bit EBCDIC equivalent.
- Requires that two ampersands or two apostrophes be specified in the character sequence for each ampersand or apostrophe required in the assembled term.

Character self-defining term	*Characters assembled*	*Hexadecimal value*	*Binary value*	
C'AB'	AB	X'C1C2'	11000001	11000010
C'C'	C	X'C3'	00000000	11000011
C'3'	3	X'F3'	00000000	11110011
C'D2'	D2	X'C4F2'	11000100	11110010
C' '	blank	X'40'	00000000	01000000
C'#'	#	X'7B'	00000000	01111011
C'&&'	&	X'50'	00000000	01010000
C' '' '	'	X'7D'	00000000	01111101
C'L' ''	L'	X'D37D'	11010011	01111101
C'5&&'	5&	X'F550'	11110101	01010000

An ASCII character self-defining term must be coded as the letter S, followed by 1 or 2 characters enclosed in apostrophes. When assembling ASCII character constants, the assembler:

- Allows any of the 256 8-bit EBCDIC characters as input. This includes the printable characters, including blanks and special characters.
- Assembles each character into its 8-bit ASCII equivalent (7 bit character code with high-order zero bit). All characters for which there is not ASCII equivalent will assemble as an ASCII blank character code (X'20').
- Requires that two ampersands or two apostrophes be specified in the character sequence for each ampersand or apostrophe required in the assembled term.

Examples:

ASCII self-defining term	Characters assembled	Hex value	Binary value	
S'AB'	AB	X'4142'	01000001	01000010
S'C'	C	X'43'	00000000	01000011
S'3'	3	X'33'	00000000	00110011
S'R3'	R3	X'5233'	01010010	00110011
S' '	blank	X'20'	00000000	00100000
S'#'	#	X'23'	00000000	00100011
S'&&'	&	X'26'	00000000	00100110
S' '' '	'	X'27'	00000000	00100111
S'I' ''	I'	X'4927'	01001001	00100111
S'7&&'	7&	X'3726'	00110111	00100110

The assembler maintains the values represented by self-defining terms as 16 bits. If a term is used as the operand on a byte immediate instruction, the low-order byte of the term is placed in the immediate field. The high-order byte must be zero.

Expressions

You can use an expression to code:

- An address
- An absolute value
- An explicit length
- A length modifier
- A duplication factor
- A complete operand

You can write an expression with a simple term or as an arithmetic combination of terms. The assembler reduces multiterm expressions to single values. Thus, you do not have to compute these values. For example, expressions are used in the following instructions as indicated:

```
          :
ADDRESS   MVW   R2,DATA+4          EXPRESSION USED AS AN ADDRESS.
VALUE     MVWI  5+2,R3               AS AN ABSOLUTE VALUE,
          :
LENGTH    DS    CL(ALPHA-BETA)       AS AN EXPLICIT LENGTH,
FACTOR    DS    (ALPHA-BETA+2)C      AS A DUPLICATION FACTOR,
OPERAND   EQU   LABEL+1              OR AS A COMPLETE OPERAND.
```

Expressions have absolute or relocatable values. Whether an expression is absolute or relocatable depends on the attributes of the terms it contains. You can use the absolute or relocatable expressions described in this section in a machine instruction or any assembler instruction other than a conditional assembly instruction. The assembler evaluates relocatable and absolute expressions at assembly time. Throughout this manual, the word "expressions" refers to these types of expressions.

Note. The three types of expressions that you can use in conditional assembly instructions are arithmetic, logical, and character. They are evaluated at preassembly time. In this manual they are always referred to by their full names; they are described in detail in Chapter 6.

An expression is *absolute* if its value is not changed by program relocation; it is *relocatable* if its value is changed upon program relocation.

Absolute Expressions

The assembler reduces an expression to a single absolute value if the expression:

- Is composed of a symbol with these values, a self-defining term, or any arithmetic combination of absolute terms
- Contains relocatable terms, alone or in combination with absolute terms, and if all these relocatable terms are paired

An expression can be absolute even though it contains relocatable terms, provided that all the relocatable terms are paired. The pairing of relocatable terms cancels the effect of relocation. The assembler reduces paired terms to single absolute terms in the intermediate stages of evaluation. The assembler considers relocatable terms as paired under the following conditions:

- The paired terms must be defined in the same control section of a source module (that is, have the same relocatability attribute).

- The paired terms must have opposite signs after all unary operators are resolved. In an expression, the paired terms do not have to be contiguous; that is, other terms can come between the paired terms.
- The value represented by the paired terms is absolute.

Register Expressions

You may code register references as expressions by following the rules for coding absolute expressions. There must be at least one register symbol present in the expression to give it the register attribute.

Note. To ensure accuracy of the cross reference listing, code the register symbol as the first term of a register expression.

The following sample code shows some relocatable and absolute terms:

```
FIRST    CSECT
         :
ABLE     DS    F                  ABLE, BAKER, CHARLIE, AND LOCREF
BAKER    DS    F                  ARE RELOCATABLE TERMS THAT CAN BE
CHARLIE  DS    F                  PAIRED IN THE SAME EXPRESSION.
         :
LOCREF   EQU   *                  LOCATION COUNTER REFERENCE
         :
ABSA     EQU   X'FFØØ'             ABSA, ABSB, AND ABSC ARE
ABSB     EQU   128                 EQUATED TO ABSOLUTE TERMS.
ABSC     EQU   C'AB'
         :
ABSD     EQU   BAKER-ABLE          ABSD AND ABSE ARE EQUATED TO
ABSE     EQU   *-CHARLIE           PAIRED RELOCATABLE TERMS.
         :
EXAMPLE1 EQU   ABSA                THE OPERANDS OF EXAMPLE1,
EXAMPLE2 EQU   15                  EXAMPLE2, EXAMPLE3, AND
EXAMPLE3 EQU   ABSA+ABSC*5         EXAMPLE4 ARE ABSOLUTE EXPRESSIONS.
EXAMPLE4 EQU   BAKER-ABLE/ABSB+ABSD
         :
SECOND   CSECT
         :
DELTA    DS    X                  DELTA, EASY, AND FOX ARE
EASY     DS    X                  RELOCATABLE TERMS THAT CAN BE
FOX      DS    X                  PAIRED IN THE SAME EXPRESSION.
         :
         END
```

Examples of valid expressions:

Paired relocatable terms	*Absolute expressions*
BAKER – ABLE	ABLE + ABSA – BAKER
CHARLIE – ABLE	DELTA – EASY + ABSC
LOCREF – CHARLIE	FOX – DELTA + BAKER – CHARLIE
DELTA – EASY	
FOX – DELTA	
Unpaired relocatable terms	*Relocatable expressions*
BAKER	BAKER + ABSA
CHARLIE	CHARLIE + X'FF'
LOCREF	FOX – 5*(BAKER – CHARLIE)
DELTA	

Relocatable Expressions

A relocatable expression is one whose value changes, for example, by a factor of 100, if the object module into which it is assembled is relocated 100 bytes away from its originally assigned storage area. The assembler reduces a relocatable expression to a single relocatable value if the expression:

- Is composed of a single relocatable term, or
- Contains relocatable terms, alone or in combination with absolute terms, and
  - All the relocatable terms but one are paired. Note that the unpaired term gives the expression a value with the relocatability attribute of that term. The paired relocatable terms and other absolute terms increase or decrease the value of the unpaired term.
  - The sign preceding the unpaired relocatable term must be positive, after all monadic operators have been resolved.

Complex Relocatable Expressions. Complex relocatable expressions, unlike relocatable expressions, can contain:

- Two or more unpaired relocatable terms, or
- An unpaired relocatable term preceded by a negative sign.

Complex relocatable expressions can be used only in A-type address constants. (See "A-type Address Constant" in Chapter 5.)

In the following sample code, EXAMPLE1, EXAMPLE2, EXAMPLE3, and EXAMPLE4 are equated to valid relocatable expressions (that is, they belong to the same control section and have the same relocatability attribute as the relocatable terms in the expressions):

```
FIRST    CSECT
         :
ABLE     DC    F'2'                ABLE, BAKER, AND CHARLIE
BAKER    DC    F'3'                ARE RELOCATABLE TERMS.
CHARLIE  DC    F'4'
         :
ABSA     EQU   1Ø                  ABSA, ABSB, AND ABSC
ABSB     EQU   *-ABLE              ARE ABSOLUTE TERMS.
ABSC     EQU   1Ø*(BAKER-ABLE)
         :
EXAMPLE1 EQU   ABLE+ABSA+1Ø        BAKER-ABLE AND CHARLIE-ABLE ARE
EXAMPLE2 EQU   BAKER-ABLE+CHARLIE  PAIRED RELOCATABLE TERMS
EXAMPLE3 EQU   BAKER+2+(CHARLIE-ABLE)
         EXTRN EXTERNAL
         :
EXAMPLE4 DC    A(ABLE-EXTERNAL)    COMPLEX RELOCATABLE EXPRESSIONS ARE
*                                  VALID IN A-TYPE ADDRESS CONSTANTS
*                                  ONLY
         :
         END
```

Rules for Coding Expressions

The rules for coding an absolute or relocatable expression are:

- An expression must not begin with an operator other than the unary minus or unary plus, and must not contain two operators in succession.
  Unary operators: −, +
  Binary operators: +, −, *, /
  Valid expressions: −2, INDEX+4
  Invalid expressions: /2, A+/2
- An expression must not contain two terms in succession. Valid expressions:
  ABLE*BAKER
  Invalid expressions: ABLEBAKER, X'FF'(10*A),C'A'B'101'
- No blanks are allowed between an operator and a term.
  Valid expression: ABLE*BAKER
  Invalid expression: ABLE * BAKER
- An expression can contain up to 16 terms and up to five levels of parentheses. Note that parentheses that are part of an operand specification do not count toward this limit (see "Parentheses in Instruction Operands" in this chapter).
- A single relocatable term is not allowed in a multiply or divide operation. Note that paired relocatable terms have absolute values and can be multiplied and divided if they are enclosed in parentheses.
- Context determines whether an asterisk (*) is the binary operator for multiplication, the location counter reference, or the indirect addressing indicator.
  Valid expressions: ABSA+*, *+3
  Invalid expressions: A*/B, ABSA+*ABSB, *3

Evaluation of Expressions

The assembler reduces a multiterm expression to a single value as follows:

- It evaluates each term.
- It performs arithmetic operations from left to right; however, multiplication and division are performed before addition and subtraction.
- In division, it gives an integer result; any fractional portion is dropped. Division by zero gives a zero result.
- Every expression is computed using 32-bit arithmetic.
- In parenthesized expressions, the assembler evaluates the innermost expressions first and then considers them as terms in the next outer level of expressions. It continues this process until the outermost expression is evaluated. It is assumed that the assembler evaluates paired relocatable terms at each level of expression nesting. The expression
  A−(X'FF'*2+B−(C/2*D)) is evaluated in the order:
  −. Evaluate C/2*D; call the result *result1*.
  − Evaluate X'FF'*2+B−*result1*; call the result *result2*.
  − Evaluate A−*result2*.
- Expression values are maintained internally to 32 bits during assembly. Negative results are carried in twos complement form, and intermediate results can range from -2^{31} through $2^{31}-1$. However, the final value of an expression must be in the allowable range for the instruction. For most instructions, that range is −65,536 through 65,535. The one exception is DC type A with length 4. That instruction initializes a doubleword value, and therefore can be in the range -2^{31} to $2^{31}-1$.

The following examples indicate the order of evaluation of expressions:

Absolute Expression	*Assumed Values*	*Value of Expression*
A + 10/B	A = 10, B = 2	15
(A + 10)/B	A = 10, B = 2	10
A/2	A = 10	5
A/2	A = 11	5
A/2	A = 1	0
10*A/2	A = 1	5
A/0	A = 1	0

Parentheses in Instruction Operands

Two types of parentheses may be used in instruction operands:

1. *Syntactic parentheses* delimit the elements of an operand. Whenever the contents of a register are to be used in an effective address calculation, that register reference is enclosed in syntactic parentheses. The following operand forms use syntactic parentheses:

```
(reg)
(reg)+
(reg)*
(reg,addr)
(reg,addr)*
disp1(reg,disp2)*
(reg,disp)*
disp(reg)*
disp(addr)*
```

   See Chapter 4 for an explanation of these forms of addressing.

2. *Arithmetic parentheses* are used to combine the terms of an arithmetic expression. They may be used in combination with syntactic parentheses subject to the following rules:
   a. Any occurrence of arithmetic parentheses must be preceded or followed by an arithmetic operator (+, −, *, /) or the indirect addressing indicator (*). The following examples show valid syntactic/arithmetic parentheses combinations:

Example	*Address Mode*
(LOC + 4)*	addr*
((R2 + 1)*2)	(reg)
(R1 + 3)/2	reg
(R2, LOC + (6*D))	(reg, addr)

   b. No operand may be completely enclosed in arithmetic parentheses. That is, the following are invalid:
      (1)((ADDRA)*) where ADDRA is an ordinary symbol and the intended addressing mode is *addr**.
      (2)((R1+4)−3) where the intended addressing mode is *reg*; the outer arithmetic parentheses will be interpreted as syntactic causing address mode *(reg)* to be generated.

Chapter 3. Functional Characteristics

Section Contents

Introduction

This chapter describes the characteristics of the IBM 4953 Processor and the IBM 4955 Processor. All information in this chapter applies to both processors, unless specifically noted otherwise. This chapter explains register usage, addressing modes, and other information you should understand to effectively use the assembler.

Registers

Each processor has one Interrupt Mask Register (IMR) and one Processor Status Word (PSW). Each priority interrupt level has eight general-purpose registers, one Instruction Address Register (IAR), one Address Key Register (AKR) (4955 Processor *only*), and one Level Status Register (LSR). All of the preceding are 16-bit registers. Optionally, each level can have installed four 64-bit floating-point registers (4955 Processor *only*).

Registers Fitted on a Per-Level Basis

Each of the four levels on the system has the following registers available to the software:

General registers (R0–R7). Also referred to simply as registers, these are eight 16-bit general-purpose registers, whose selection is controlled by the R fields in instructions.

Floating-point registers (FR0–FR3) (4955 Processor *only*). Four 64-bit floating-point registers are provided with the floating-point optional feature. They are selected by the R fields in floating-point instructions.

Instruction Address Register (IAR). The IAR contains the address of the leftmost byte of the next instruction to be executed.

Address Key Register (AKR) (4955 Processor *only*). This 16-bit register contains three address keys and an address key control bit associated with address space management and the storage protection mechanism. Separate 3-bit fields contain an address key for (1) instruction address space, (2) operand1 address space, and (3) operand2 address space. For more information, see Chapter 8 of *IBM 4955 Processor and Processor Features Description*, GA34-0021.

Bits	*Contents*
00	Equate operand spaces
01	Not used, always zero
02	Not used, always zero
03	Not used, always zero
04	Not used, always zero
05	Operand 1 key (bit 0)
06	Operand 1 key (bit 1)
07	Operand 1 key (bit 2)
08	Not used, always zero
09	Operand 2 key (bit 0)
10	Operand 2 key (bit 1)
11	Operand 2 key (bit 2)
12	Not used, always zero
13	Instruction space key (bit 0)
14	Instruction space key (bit 1)
15	Instruction space key (bit 2)

Level Status Register (LSR). This 16-bit register contains information about the status of an interrupt level. It has this format:

Bit	*Contents*
00	Even indicator
01	Carry indicator
02	Overflow indicator
03	Negative result indicator
04	Zero result indicator
05	Not used, always zero
06	Not used, always zero
07	Not used, always zero
08	Supervisor state
09	In process
10	Trace
11	Summary mask
12	Not used, always zero
13	Not used, always zero
14	Not used, always zero
15	Not used, always zero

Bits not used in the LSR are always zero.

Registers Fitted on a Per-System Basis

The registers discussed in this section are addressable through assembler-language instructions.

Interrupt Mask Register (IMR). A 16-bit register used for control of interrupts. Bit zero controls level 0, bit one controls level 1, and so on. A one in bit position N enables interrupts on level N, while a zero disables level N.

Processor Status Word (PSW). The PSW is a 16-bit register that reports the specific condition that caused an exception interrupt (program check, machine check, or soft exception check).

The PSW contains the following:

Type of Interrupt	*Bit*	*Meaning*
Program check	00 01 02 03	Specification check Invalid storage address Privilege violate Protect check
Either program check or soft exception trap	04	Invalid function
Soft exception trap	05 06 07	Floating-point exception Stack exception Not used
Machine check	08 09 10 11	Storage parity check Not used CPU control check I/O check
Status flags	12 13 14	Sequence indicator Auto-IPL Translator enabled
Power/Thermal	15	Power/Thermal warning

Bits not used in the PSW are always zero.

Console Data Buffer. A 16-bit register that is accessible with the full-function console. Issue the CPCON instruction to read this buffer.

Number Representation

Operands can be signed or unsigned. An unsigned number is a binary integer in which all bits contribute to its magnitude. A storage address is an example of an unsigned number. Signed positive numbers are represented in true binary notation with the sign bit (high-order) set to zero. Signed negative numbers are represented in twos complement notation with a one in the sign bit. To get the twos complement of a number, invert each bit of the number and add a one to the low-order bit position.

When the number is positive, all bits to the left of the most significant bit of the number, including the sign bit, are zero. When the number is negative, all bits to the left of the most significant bit of the number, including the sign bit, are set to one.

Twos complement notation does not include a negative zero. The maximum positive number consists of an all-one integer field with a sign bit of zero. The maximum negative number (the negative number with the greatest absolute value) consists of an all-zero integer field with a one-bit for sign.

Indicators

A single set of add and subtract integer arithmetic operations performs both signed arithmetic and unsigned (that is, binary or logical) arithmetic. The carry and overflow indicators are set to reflect the result in both cases.

For signed addition and subtraction, the overflow indicator signals a result that exceeds the representation capability of the system. When an overflow occurs, the carry indicator and the contents of the result operand together form a valid result of which the carry indicator is the sign bit for addition and the *complement*

of the sign bit for subtraction. If there is no overflow, the carry indicator contains no information about the result.

For unsigned addition and subtraction, the carry indicator signals that:

- On an add instruction, a carry out of the high-order bit position has occurred
- On a subtract operation, a borrow beyond the high-order bit position has occurred

When a carry is indicated on an add operation, the carry indicator and the result operand together form a valid result of which the carry indicator is the most significant bit.

When a borrow is signaled on a subtract operation, the result is in twos complement form. The overflow indicator contains no information about unsigned addition or subtraction operations.

The following example shows how the processor performs the subtraction, with respect to the setting of the carry indicator.

Add Complement (Subtract)

```
 +6          0110
-(+7)        1001
             1111 ──────► No carry out, carry indicator on
```

If the same operation were to be done with the binary subtract method, we see that a borrow out would occur:

Binary Subtract

```
          0   1 10 10
          1  10  0  0 10
 +6         0  1  0  0
-(+7)       0  1  1  1
      ◄──── 1  1  1  1 ◄──── Borrow out of Bit 0,
                             Carry indicator on
```

The hardware adds or subtracts a negative and positive number by using the add complement method. The operations are identical in the hardware, except that the carry indicator settings are different for add and subtract.

The following examples show how the hardware adds and subtracts numbers that produce the same result:

Add

```
 +3    0011
+(-4)  1100
       1111 ──────► No carry out, carry indicator off
```

Subtract

```
 +3    0011
-(+4)  1100
       1111 ──────► No carry out, carry indicator on
```

The following examples show how the hardware adds and subtracts, with respect to the overflow indicators. The processor recognizes an overflow condition by observing the internal carries both into and out of the high-order bit position (the sign bit). If the carries disagree, an overflow condition exists. If they agree, there is no overflow. There are four possibilities:

- No carry in and no carry out (carries agree—no overflow)
- Both carry in and carry out (carries agree—no overflow)
- Carry in, but no carry out (carries disagree—overflow)
- Carry out, but no carry in (carries disagree—overflow)

The four possible cases are shown in the following examples. Decimal equivalents are given for comparison.

Example 1 – No Overflow

```
          111  11  <------------ Carries
+62       0011 1110
+27       0001 1011
+ 89      0101 1001
```

In this straightforward addition, there is neither a carry into the high-order bit position nor a carry out; therefore, there is no overflow. The carry indicator is off.

Example 2 – No Overflow

```
        1  1111  1  <------------ Carries
+62        0011 1110
-27        1110 0101
+35        0010 0011
```

Here there is both a carry in and a carry out; hence, no overflow. The carry indicator is on.

Example 3 – Overflow

```
           1111  <------------ Carries
 +62       0011 1110
 +89       0101 1001
+151       1001 0111
```

Here there is a carry into the high-order position, but no carry out. Since the overflow indicator is on, the result has exceeded the capacity of the system. The result is an unsigned binary integer, and the carry indicator contains the sign bit. The overflow condition is also evident from the decimal result 151, which exceeds 127, the maximum value that can be represented in eight bits.

Example 4 – Overflow

```
          1       11  <------------ Carries
 -62        1100 0010
 -89        1010 0111
-151        0110 1001
```

In this example, there is a carry out of the high-order bit position, but no carry in. This causes an overflow condition. The carry indicator is on. As in the previous example, the result is an unsigned binary integer with the carry indicator containing the sign bit. Again, the presence of overflow is evident from the decimal result.

Other Uses of Indicators

The even, carry and overflow indicators contain the condition code following an I/O instruction or interrupt. The even indicator is bit 0 of the condition code, the carry indicator is bit 1, and the overflow indicator is bit 2. For detailed information about this use of indicators, refer to the processor description manual for your processor.

The carry indicator is also used to reflect the value of the last bit shifted out of the target register on shift left logical operations. The overflow indicator is used in these operations to indicate whether bit 0 of the shifted register has changed (if bit 0 has changed, the sign of the number has changed).

The carry indicator is used on shift left logical and count operations to reflect the value of the last bit shifted out of the register.

A compare operation affects the indicators in the same manner as a subtract operation. Compare instructions are usually used in conjunction with conditional branches or jumps. The specified conditions in conditional branches and jumps are named with respect to the indicators, so that in all compare instructions the subtracted-from operand is compared relative to the other operand. For example, in a Compare Word instruction (CW R1,R2) where the contents of R1 are subtracted from the contents of R2, the indicators reflect *arithmetically less than* if the contents of R2 are arithmetically less than the contents of R1.

The indicators are tested according to a selected condition on a conditional branch or jump instruction. For a discussion of the conditions set by these indicators, see "Using Compare Instructions" in Chapter 4.

Multiplication and division always operate on signed numbers. The indicator settings for these operations are described in Chapter 4.

Since the complement of the maximum representable negative number is itself not representable, an attempt to complement this number turns on the overflow indicator.

The result indicators are the zero, negative, and even indicators. A positive result is indicated when the zero and negative indicators are both zero. These indicators reflect the result of the last arithmetic or logical operation performed. See the individual instruction descriptions in Chapter 4 for details of indicator setting.

All indicators are changed by the data associated with the Set Indicators (SEIND) and Set LSB (SELB) instructions.

Indicators (carry, overflow, zero, negative and even) are set or reset at the end of each floating-point instruction. Whether each is set or reset is described in the detailed instruction descriptions in Chapter 4.

Storage Addressing

All storage addresses are 16-bit, unsigned, binary integers. The direct address range of the system is 64KB. The addressable unit of main storage is the byte, and all references to storage locations are byte addresses. Instructions refer to bits, bytes, words, doublewords, or fields as data types. Some rules concerning storage addressing are:

- All instructions must start on an even byte boundary. The effective address for all branching instructions must be on an even byte boundary.
- All word and doubleword operand addresses must be on an even byte boundary.
- In the case of indirect addressing, the address operand must be on an even byte boundary.
- A stack control block must be on an even byte boundary.
- All byte, word, and doubleword operand addresses point to the leftmost byte in the operand.
- All bit and field addresses are specified by a byte address and a bit displacement, and point to the leftmost bit in a field.
- In order to provide maximum addressing range, some instructions refer to an even byte displacement that is added to the contents of a register. In these cases, the register must also contain an even byte address to point to a word or doubleword operand.

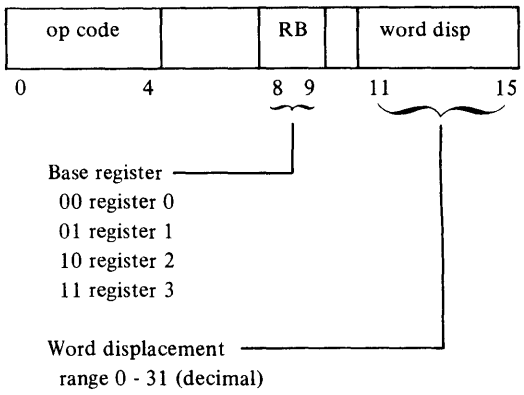
If one of the above rules is violated, a program check interrupt occurs with *specification check* set in the PSW. The instruction is suppressed.

Effective Address Generation

For purposes of storage efficiency, certain instructions formulate storage operand addresses in a specialized manner. These instructions have self-contained fields that the assembler uses when generating effective addresses. Standard methods for deriving effective addresses are described in this section.

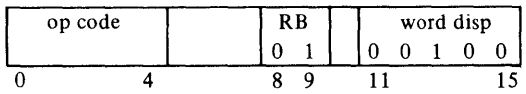
Base Register, Word Displacement Short

Instruction format:



The 5-bit unsigned integer *word disp* is doubled in magnitude to form a byte displacement, then is added to the contents of the specified base register to form the effective address.

Example:

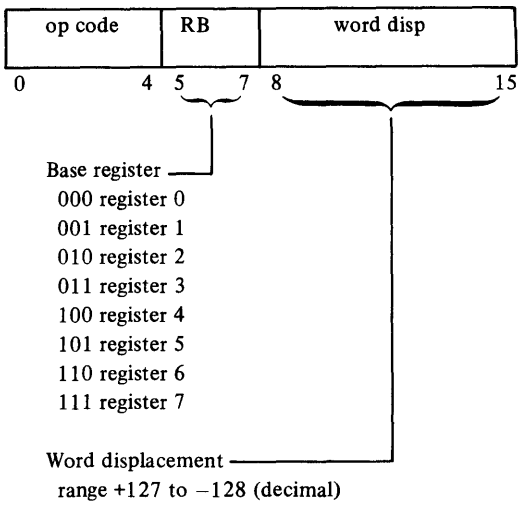


		(HEX)
Contents of register 1 (RB)	0000 0000 0110 0000	0060
Word displacement doubled	+ 0 1000	8
Effective address	0000 0000 0110 1000	0068

This is coded as *shortaddr* in the MVWS instruction.

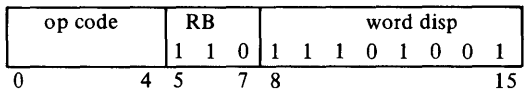
Base Register, Word Displacement

Instruction format:



The 8-bit signed integer *word disp* is doubled in magnitude to form a byte displacement, then is added to the contents of the specified base register to form the effective address. The word displacement can be either positive or negative; bit 8 of the instruction word is the sign bit for the displacement value. If this high-order bit of the displacement field is a 0, the displacement is positive with a maximum value of +127 (decimal). If the high-order bit of the displacement field is a 1, the displacement is negative with a maximum value of −128. A negative displacement is represented in twos complement form.

Example:



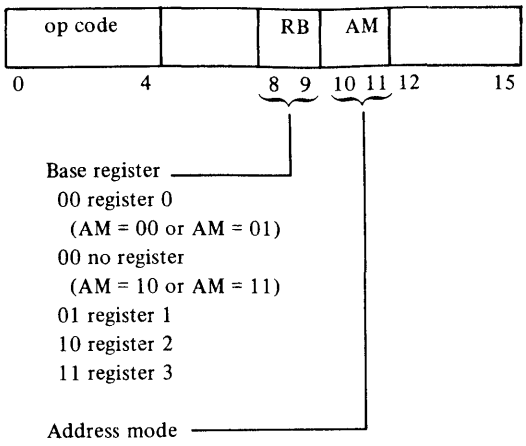
Note: This example shows a negative word displacement (−17 hex) shown in twos complement.

			(HEX)
Contents of register 6 (RB)		0000 0000 1000 0110	0086
Word displacement doubled (Sign bit is propagated left)	+	1111 1111 1101 0010	− 2E
Effective address		0000 0000 0101 1000	0058

This is coded as *(reg,jdisp)* in the BXS instruction.

Four-bit Address Argument

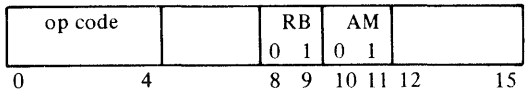
Instruction format:



The address mode has the following significance:
AM=00. The contents of the selected base register form the effective address. The equivalent assembler instruction operand format is $reg^{0\text{-}3}$.
AM=01. The contents of the selected base register form the effective address. After use, the base register contents are increased by the number of bytes in the operand.

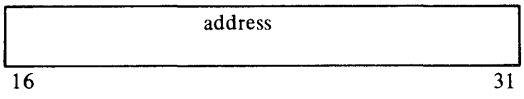
For certain instructions (SELB, CPLB, LMB, STM, PB, PD, PW, PSB, PSD, and PSW) the effective address does not point to an operand, but to a control block or other system parameters. For these instructions, the contents of the specified register are increased by two.

Example:



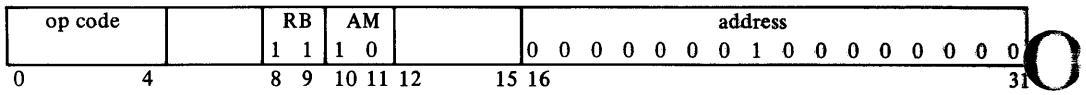
		(HEX)
Effective address (Contents of register 1)	0000 0000 1000 0000	0080
Contents of register 1 after instruction execution		
Byte operand	0000 0000 1000 0001	0081
Word operand	0000 0000 1000 0010	0082
Doubleword operand	0000 0000 1000 0100	0084

The equivalent assembler instruction operand format is $(reg^{0\text{-}3})+$.
AM=10. An additional word is appended to the instruction. The word has the following format.



- If RB is zero, the appended word contains the effective address.
- If RB is not zero, the contents of the selected base register and the contents of the appended word are added to form the effective address.

Example:

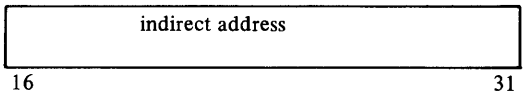


		(HEX)
Contents of register 3	0000 1000 0000 0000	0800
Contents of appended word	+0000 0001 0000 0000	0100
Effective address	0000 1001 0000 0000	0900

The equivalent assembler instruction operand formats are: *addr* and *(reg*[13]*,waddr).*

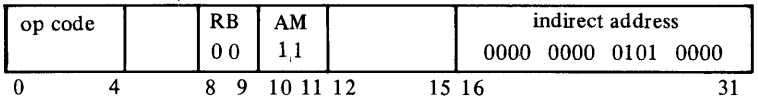
AM=11. An additional word is appended to the instruction.

- If RB is zero, the appended word has the format:



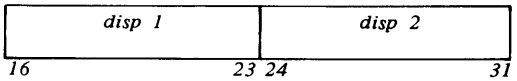
This address points to a main storage location, on an even byte boundary, that contains the effective address.

Example:



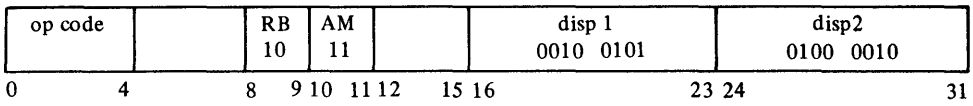
					(HEX)
Contents of appended word	0000	0000	0101	0000	0050
Effective address equals contents of storage at address 0050 (hexadecimal)	0000	0100	0000	0000	0400

- If RB is not zero, the appended word has the format:



The two displacements are unsigned 8-bit integers. Displacement 2 is added to the contents of the selected base register to generate a main storage address. The contents of this storage location are added to displacement 1, resulting in the effective address.

Example:

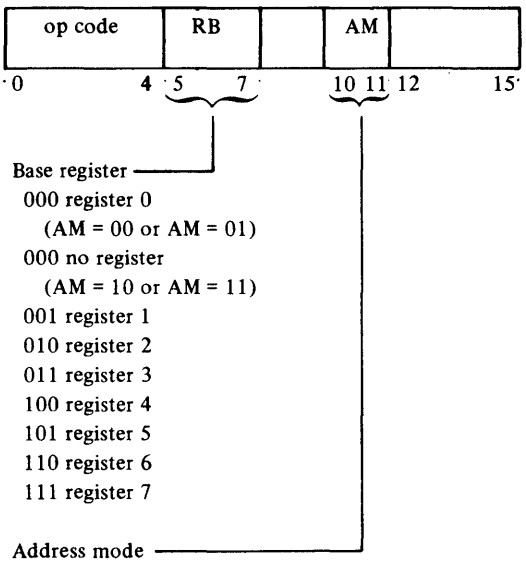


		(HEX)
Contents of register 2	0000 0101 0011 0101	0535
Displacement 2	+ 0100 0010	42
Storage address	0000 0101 0111 0111	0577
Contents of storage at address 0577 (Hexadecimal)	0000 0100 0001 0000	0410
Displacement 1	+ 0010 0101	25
Effective address	0000 0100 0011 0101	0435

The equivalent assembler instruction operand formats are:
disp1(reg 1-3,disp2)
*disp(reg 1-3)**
*(reg 1-3)**
*(reg 1-3,disp)**.

Five-bit Address Argument

Instruction format:

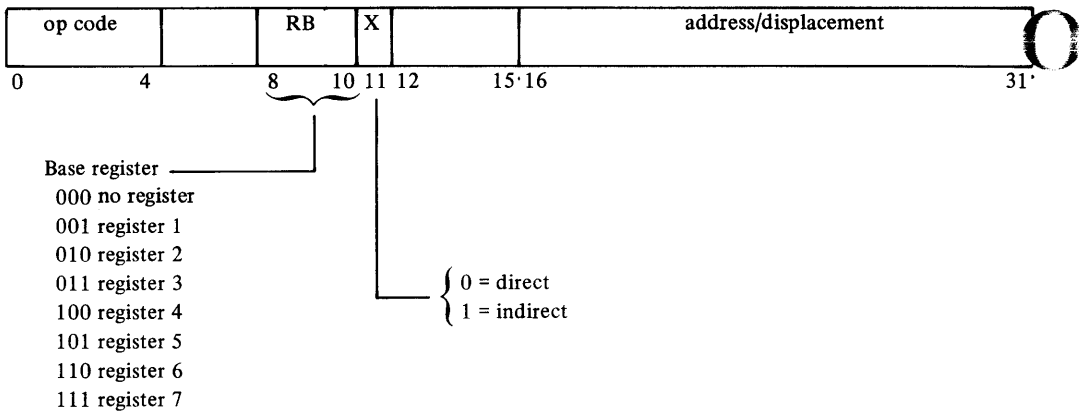


Operation of this mode is identical to the four-bit argument, but provides additional base registers.

This is coded as *addr5*.

Base Register, Storage Address

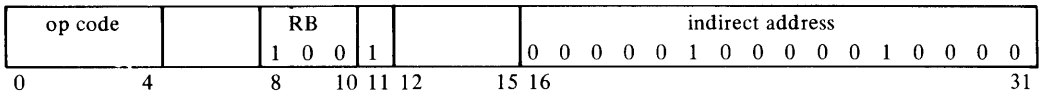
Instruction format:



- If RB is zero, the address field contains the effective address.
- If RB is not zero, the contents of the selected base register and the contents of the address field are added together to form the effective address.

Note. Bit 11, if a one, specifies that the effective addressing is indirect.

Example:



		(HEX)
Contents of register 4	0000 0001 0000 0000	0100
Address field	+ 0000 0100 0001 0000	0410
Storage address	0000 0101 0001 0000	0510
Effective address		
Contents of storage at address 0510 (hexadecimal)	0000 0110 0100 0000	0640

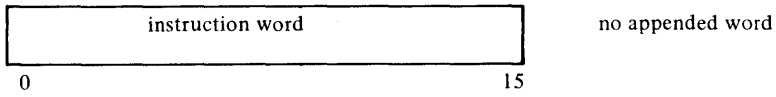
This is coded as *longaddr*.

Instruction Length Variations for Address Arguments

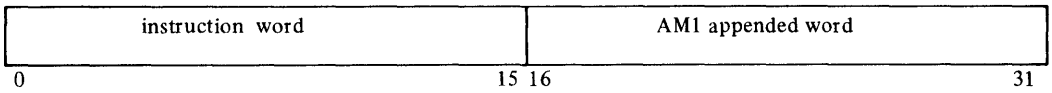
- One-word instructions that contain a single AM field become two words in length if AM is equal to 10 or 11. The appended word follows the instruction word.
- Two-word instructions that contain a single AM field become three words in length if AM is equal to 10 or 11. The AM word is appended to the first instruction word. The *data* or *immediate* field then becomes the third word of the instruction.
- One-word instructions that contain two AM fields (AM1 and AM2) are one, two, or three words in length depending on the values of AM1 and AM2. The AM1 word is appended first, then the AM2 word.

Examples:

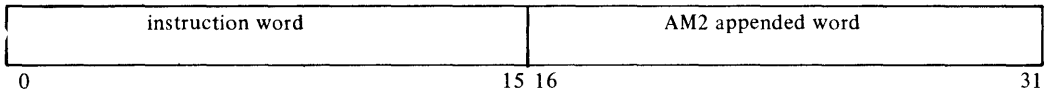
- AM1=00 or AM1=01; AM2=00 or AM2=01



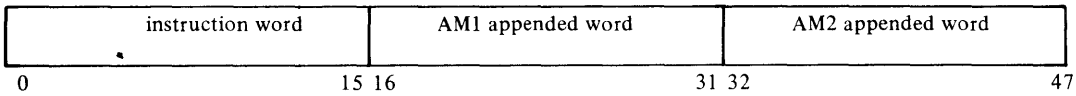
- AM1=10 or AM1=11; AM2=00 or AM2=01



- AM1=00 or AM1=01; AM2=10 or AM2=11



- AM1=10 or AM1=11; AM2=10 or AM2=11



Stack Operations

Stack Control Block

Stacking is a simple efficient mechanism for queuing data and/or parameters. Basically, a stack is a LIFO queue. There are operations that push a data item or parameter into the stack and operations that pop the top item from the stack. In addition, there are limit-checking facilities, which test for overflow and underflow of a stack area.

Any contiguous area of storage can be defined as a stack. Each logical stack is defined by a stack control block in the following format:

Word 1 Top element address (TEA)
Word 2 High limit address of stack (HLA)
Word 3 Low limit address of stack (LLA)

The size of the stack is equal to HLA minus LLA. When a stack is empty, the top element address is equal to the high limit address. The HLA must be greater than the LLA.

When an item is pushed to the stack, the address value in the TEA is decreased and compared against the LLA. If it is less than the LLA, a stack overflow exists. Stack overflow causes a soft exception interrupt to occur, with stack exception set in the PSW. The TEA is unchanged. If the stack does not overflow, the TEA is updated and the data item is moved to the storage location defined by the TEA.

When an item is popped from a stack, the TEA is compared against the HLA. If it is greater than or equal to the HLA, a stack underflow exists. Stack underflow causes a soft exception interrupt to occur, with the stack exception bit on in the PSW. If the stack does not underflow, the data item defined by the current TEA is moved to a specified register and the address value of the TEA is increased.

Note. It is possible to pop data from beyond the stack boundary if the TEA is less than the HLA, and the operand size is greater than (HLA-TEA).

Stack operations are register-to-storage for push, and storage-to-register for pop. Bytes, words, doublewords, and register blocks can be stacked. You are responsible for ensuring that the TEA word of a stack control block contains an even byte address for word, doubleword, and register blocks operations. All stack control blocks must be aligned on a word boundary.

Linkage Stacking

A word stack can be used for subroutine linkage, as a method of saving/restoring caller status and allocating dynamic work areas. The STM/LMB instruction pair operate using a stack area. The STM instruction specifies:

- Stack control block address (A)
- Limit register number (R)
- Number of bytes to allocate as dynamic work area (N)

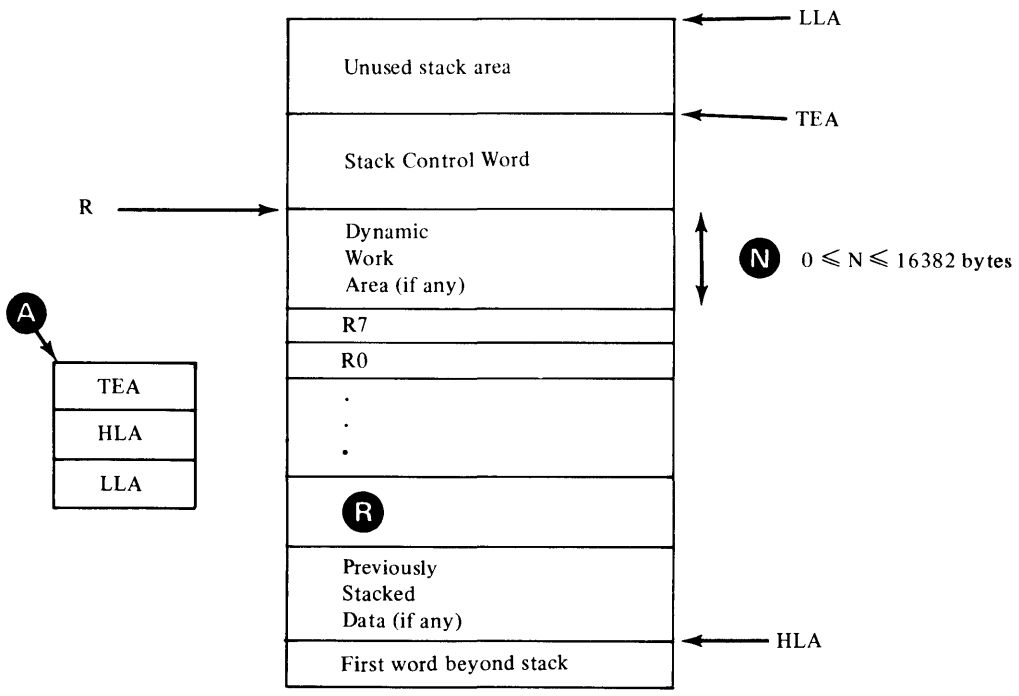
Note. You code this value in bytes, then the assembler converts it to words for use by the hardware instruction.

When STM is executed, the TEA value is decreased by the size of the area allocated on the stack before an overflow check is made. The size of that area is N, plus two bytes for each register saved, plus two bytes for the control word. The link register (R7) and register 0 through register R are saved sequentially in the stack area and the address of the dynamic work area is returned to you in register R. If R7 is specified, only R7 is stored. The value of R and N are also saved as an entry in the stack. When an LMB instruction is executed, these values (R, N) are retrieved from the stack and are used to control the reloading of registers and restoring the stack control pointer to its former status. The contents of R7 are then loaded into the IAR, returning to the calling routine. For example, if you want to store register 7, then registers 0 through R, in a stack defined by the stack control block at location A, *and* you want to allocate N bytes of dynamic work area:

```
STM R,A,N
```

this is how the stack is stored:

Low Storage.



High Storage

Chapter 4. Machine Instructions

Section Contents

Hardware instructions are represented symbolically by assembler-language statements. Each statement generates one hardware instruction—the actual instruction generated depends on the operation code and the syntax of the operand.

Each mnemonic operation code specifies the function of an instruction and the type of data it operates on. For example, the Move Word (MVW) instruction moves (MV) a word (W) from a register to storage, storage to a register, storage to storage, or a register to a register, depending on the operands you code. Based on the syntax of the operands, the assembler generates one of several possible hardware instructions. If more than one hardware instruction can perform the operation specified by the mnemonic and its operand, the assembler generates the one that is most efficient in timing and storage usage.

This chapter discusses the assembler-language machine instructions—how you code them and what they do.

Coding Notes

- Data flow, when it modifies a field, is always from left to right.
- Registers used in effective address calculations are always in parentheses.
- An address specification followed by an asterisk indicates indirect addressing. Here, the effective address is the contents of the addressed storage location.
- The *(reg)+* format indicates that, after use, the contents of the *reg* are increased by the number of bytes addressed.

Coding The Assembler Language Instructions

This section explains the symbols that are used as generalized operands in the discussion of machine instructions. (The discussion of machine instructions comprises the remainder of this chapter.)

abcnt

An absolute value or expression representing the size of a work storage area to be allocated by the Store Multiple (STM) instruction. The value you code must be an even number in the range 0–16382.

addr

An address value. Code an absolute or relocatable expression in the range 0–65535.

addr4

An address value that you code in one of the following forms:

(reg $^{0-3}$*)*

The effective address is the contents of the register *reg* $^{0-3}$.

(reg $^{0-3}$*)+*

The effective address is the contents of the register *reg* $^{0-3}$. After an instruction uses it, the contents of the register are increased by the number of bytes addressed by the instruction.

addr

The effective address is the value of *addr*, unless the instruction is in the domain of a USING directive and *addr* is in the range of the same USING directive. If they are, the assembler computes the effective address as a displacement (−32768 to +32767 or 0 to 65535) from the base register, which must be *reg* $^{1-3}$.

*addr**

The effective address is the contents of storage at the address defined by *addr*, unless the instruction is in the domain of a USING directive and *addr* is in the range of the same USING directive. If they are, the assembler computes the effective address as the contents of storage at the address defined by a displacement (0–255) from the base register, which must be *reg*[1-3].

*disp(addr)**

If the instruction is in the domain of a USING directive and the operand is in the range of this same USING directive, then the assembler will compute the displacement and register combination which will reference the requested location; i.e., the resulting addressing mode will be disp[1](reg[1-3],disp[2])*

(reg[1-3]*,waddr)*

The effective address is the contents of the register *reg*[1-3], added to the value of *waddr*.

disp1(reg[1-3]*,disp2)**

The effective address is calculated as follow: The contents of the register *reg*[1-3] are added to the value of the displacement *disp2* to form an address. The contents of that storage location are added to the value of *disp1* to form the effective address.

disp(reg[1-3]*)**

The effective address is the contents of storage at the address defined by the contents of *reg*[1-3], added to the value of *disp*.

(reg[1-3]*)**

The effective address is the contents of storage at the address defined by the contents of *reg*[1-3].

(reg[1-3]*,disp)**

The contents of *reg*[1-3] are added to *disp*, forming an address. The contents of storage at that address form the effective address.

Note. For the byte addressing, the effective address can be even or odd. For word or doubleword addressing, the effective address must be even.

addr5

An address value that you code in one of the following forms:

(reg)

The effective address is the contents of the register *reg*.

(reg)+

The effective address is the contents of the register *reg*. After an instruction uses it, the contents of the register are increased by the number of bytes addressed by the instruction.

addr

The effective address is the value of *addr*, unless the instruction is in the domain of a USING directive and *addr* in the range of the same USING directive. If they are, the assembler computes the effective address as a displacement (−32768 to +32767 or 0 to 65535) from the base register, which must be *reg*[1-7].

*addr**

The effective address is the contents of storage at the address defined by *addr*, unless the instruction is in the domain of a USING directive and *addr* is in the range of the same USING directive. If they are, the assembler computes the effective address as the contents of storage at the address defined by a displacement (0–255) from the base register, which must be *reg*[1-7].

*disp(addr)**

If the instruction is in the domain of a USING directive and

the operand is in the range of this same USING directive, then the assembler will compute the displacement and register combination which will reference the requested location; i.e., the resulting addressing mode will be $disp^1(reg^{1-7},disp^2)$*

(reg 1-7,waddr)

The effective address is the contents of *reg* 1-7, added to the value of *waddr*.

*disp1(reg 1-7,disp2)**

The effective address is calculated as follows: contents of the register *reg* 1-7 are added to the value of the displacement *disp2* to form an address. The contents of that storage location are added to the value of *disp1* to form the effective address.

*disp(reg 1-7)**

The effective address is the contents of storage at the address defined by the contents of *reg* 1-7, added to the value of *disp*.

*(reg 1-7)**

The effective address is the contents of storage at the address defined by the contents of *reg* 1-7.

*(reg 1-7,disp)**

The contents of *reg* 1-7 are added to *disp*, forming an address. The contents of storage at that address form the effective address.

Note. For byte addressing, the effective address can be even or odd. For word or doubleword addressing, the effective address must be even.

bitdisp

A displacement into a bit field. Code an absolute value or expression in the range 0–63.

byte

A byte value. Code an absolute value or expression in the range −128 to +127 or 0 to 255.

cnt16

A single word (one register) shift count. Code an absolute value or expression in the range 0–16.

cnt31

A doubleword (register pair) shift count. Code an absolute value or expression in the range 0–31.

cond

A condition code value. Code an absolute value or expression in the range 0–7.

disp

A byte address displacement. Code an absolute value or expression in the range 0–255.

freg

A floating-point register. Code either a predefined floating register symbol (FR0–FR3) or a symbol that is equated to the desired register number (0, 1, 2, or 3). Symbols are equated with EQUR statements, which must precede the instruction using the register symbol.

jaddr

The address of an instruction that is within −256 to +254 bytes of the byte following a jump instruction. Code a relocatable expression.

jdisp

A byte address displacement. Code an even absolute value or expression in the range −256 to +254.

longaddr

An address value that you code in one of the following forms:

addr

The effective address is the value of *addr*, unless the instruction is in the domain of a USING directive and *addr* is in the range of the same USING

directive. If they are, the assembler computes the effective address as a displacement (−32768 to +32767 or 0 to 65535) from the base register, which must be *reg*[1-7].

*addr**

The effective address is the contents of storage at the address defined by *addr*, unless the instruction is in the domain of a USING directive and *addr* is the range of the same USING directive. If they are, the assembler computes the effective address as the contents of storage at the address defined by a displacement (−32768 to +32767 or 0 to 65535) from the base register, which must be *reg*[1-7].

(reg[1-7]*,waddr)*

The effective address is the contents of *reg*[1-7], added to the value of *waddr*.

(reg[1-7]*,waddr)**

The contents of *reg*[1-7], plus *waddr*, form an address. The contents of storage at that location form the effective address.

(reg[1-7]*)*

The effective address is the contents of the register *reg*[1-7].

(reg[1-7]*)**

The effective address is the contents of storage at the address defined by the contents of *reg*[1-7].

raddr

An address value. Code a relocatable expression in the range 0–65535.

reg

A general-purpose register. Code either a predefined register symbol (R0–R7) or a symbol that is equated to the desired register number (0, 1, 2, 3, 4, 5, 6, or 7). Symbols are equated with EQUR statements, which must precede the instruction using the register symbol.

reg[0-3]

A general-purpose register. Code either a predefined register symbol (R0–R3) or a symbol that is equated to the desired register number (0, 1, 2, or 3). Symbols are equated with EQUR statements, which must precede the instruction using the register symbol.

reg[1-3]

A general-purpose register. Code either a predefined register symbol (R1–R3) or a symbol that is equated to the desired register number (1, 2, or 3). Symbols are equated with EQUR statements, which must precede the instruction using the register symbol.

reg[1-7]

A general-purpose register. Code either a predefined register symbol (R1–R7) or a symbol that is equated to the desired register number (1, 2, 3, 4, 5, 6, or 7). Symbols are equated with EQUR statements, which must precede the instructions using the register symbol.

ubyte

An unsigned byte value or mask. Code an absolute value or expression in the range 0–255.

vcon

An ordinary symbol that is defined externally from the current source program.

waddr

A one-word address value. Code an absolute or relocatable expression in the range −32768 to +32767 or 0 to 65535.

wdisp

An even byte address displacement. Code an absolute value or expression in the range 0–62.

word

A word value. Code an absolute value or expression in the range −32768 to +32767 or 0 to 65535.

Data Movement Instructions

Fill Byte Field and Decrement (FFD)

This instruction fills a field in storage, right-to-left, with a byte from a register.

Name	*Operation*	*Operand*
[label]	FFD	reg, (reg)

Here is how to use FFD:

1. Before FFD, code an instruction to load register 7 with the size (in bytes) of the destination field. Note that this is an unsigned value.
2. For *reg*, code the register from which the byte is moved.
3. For *(reg)*, code the address of the rightmost byte of the destination field.

Here is what FFD does:

1. It moves bits 8–15 of *reg* to the rightmost byte of the *(reg)* field.
2. It then moves the same byte from *reg* to the byte (in the *(reg)* field) to the left of the preceding byte.
3. It proceeds to the left, moving one byte at a time from *reg* to *(reg)*, until it has moved the number of bytes specified by register 7.

When FFD is finished, register 7 contains 0, *reg* is unchanged, and *(reg)* contains the address of the byte before the leftmost byte in the field. That is, if the leftmost byte is 0207, *(reg)* points to 0206.

Indicators

The carry and overflow indicators are unchanged. The remaining indicators reflect the last byte moved.

FFD Example

```
FFD R5,(R6)
```

Assume that:

- Register 7 contains X'0003'—the number of bytes to be moved,
- Register 5 contains X'34A7', and
- Register 6 contains X'0300'—the address of the rightmost byte in the destination field.

As Figure 4-1 shows, FFD moves the value A7 into:

(1) Byte 0300,
(2) Byte 02FF, then
(3) Byte 02FE.

After FFD:

- Register 7 contains 0,
- Register 5 contains X'34A7,' and
- Register 6 contains X'02FD'.

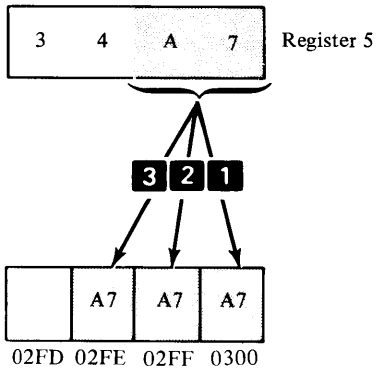


Figure 4-1. FFD example.

Coding Hint

Use this instruction when you want to clear an area—that is, fill it with blanks or zeros.

Fill Byte Field and Increment (FFN)

This instruction fills a field in storage, left-to-right, with a byte from a register.

Name	*Operation*	*Operand*
[label]	FFN	reg, (reg)

Code FFN like FFD, with one difference. For *(reg)*, code the address of the leftmost byte of the destination field.

FFN does the same thing as FFD, with one exception. The *reg* byte moves to the *leftmost* byte of the *(reg)* field, and proceeds to the *right*.

When FFN is finished, register 7 contains 0, *reg* is unchanged, and *(reg)* contains the address of the byte following the rightmost byte in the field. That is, if the rightmost byte is 010B, *(reg)* points to 010C.

Indicators

The carry and overflow indicators are unchanged. The remaining indicators reflect the last byte moved.

FFN Example

```
FFN R5,(R6)
```

Assume that:

- Register 7 contains X'0003'—the number of bytes to be moved,
- Register 5 contains X'34A7', and
- Register 6 contains X'0600'—the address of the leftmost byte in the destination field.

As Figure 4-2 shows, FFN moves the value A7 into:

(1) Byte 0600,
(2) Byte 0601, then
(3) Byte 0602.

After FFN:

- Register 7 contains 0,
- Register 5 contains X'34A7', and

- Register 6 contains X'0603'.

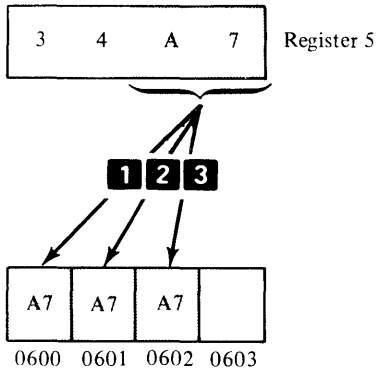


Figure 4-2. FFN example.

Coding Hint

Use this instruction when you want to clear an area—that is, fill it with blanks or zeros.

Interchange Registers (IR)

This instruction interchanges the contents of two registers.

Name	*Operation*	*Operand*
[label]	IR	reg, reg

Indicators

The indicators are set to reflect the new contents of the register defined by the second operand.

IR Example

```
IR R4,R1
```

Assume that register 4 contains X'1234' and register 1 contains X'5678'. After IR, register 4 contains X'5678' and register 1 contains X'1234'.

Move Address (MVA)

This instruction places an effective address into a register or a storage location.

Name	*Operation*	*Operand*
[label]	MVA	addr4, reg addr, addr4

Note. The *addr* or *addr** form of the first operand *must* be coded as a relocatable expression.

Indicators

The carry and overflow indicators are unchanged. The remaining indicators reflect the new contents of the second operand.

MVA Example

```
MVA LOC1,R3
```

This instruction loads the address of LOC1 into register 3.

Move Byte (MVB)

This instruction moves one byte from a register to storage, from storage to a register, or from storage to storage.

Name	*Operation*	*Operand*
[label]	MVB	reg, addr4 addr4, reg addr5, addr4

- In the register-to-storage format, bits 8–15 of *reg* are moved to *addr4*.
- In the storage-to-register format, the byte is moved from *addr4* to bits 8–15 of the register. The high-order bit of the moved byte is propagated through bits 0–7 of *reg*.
- In the storage-to-storage format, the byte moves from *addr5* to *addr4*.

Indicators

The carry and overflow indicators are unchanged. The remaining indicators reflect the byte moved.

MVB Examples

```
MVB R3,(R2)
```

Bits 8–15 of register 3 are moved to the storage location whose address is in register 2.

```
MVB 6(R3,4)*,R5
```

- The contents of register 3, plus 4, form an address.
- The contents of that storage location, plus 6, form the address of the byte to be moved.
- MVB moves the byte to bits 8–15 of register 5.
- If the high-order bit of the moved byte is 0, bits 0–7 of register 5 contain zeros; if the high-order bit is 1, bits 0–7 contain ones.

```
MVB THERE,HERE+1
```

The first byte of storage location THERE is moved to one byte past storage location HERE.

Move Byte and Zero (MVBZ)

This instruction moves a byte from storage to a register, then replaces the byte in storage with zeros.

Name	*Operation*	*Operand*
[label]	MVBZ	addr4, reg

MVBZ moves a byte from *addr4* to bits 8–15 of *reg*. The high-order bit of the moved byte is propagated through bits 0–7 of the register.

After the move, the byte at *addr4* is filled with zeros.

Indicators

The carry and overflow indicators are unchanged. The remaining indicators reflect the moved byte.

MVBZ Example

```
MVBZ LOC4,R3
```

MVBZ moves the contents of the byte at LOC4 to bits 8–15 of register 3, and propagates the high-order bit of the byte through bits 0–7 of the register. After MVBZ executes, LOC4 contains zeros.

Move Byte Field and Decrement (MVFD)

This instruction moves a specified number of bytes, one byte at a time, right-to-left, from one storage location to another.

Name	*Operation*	*Operand*
[label]	MVFD	(reg), (reg)

MVFD moves a field between two storage locations. For *(reg),(reg)* code the registers that contain the addresses of the rightmost bytes of the source and destination fields.

MVFD assumes that you have loaded register 7 with an unsigned number—the number of bytes to be moved. If R7 contains zero, this instruction is treated as a no-operation.

After MVFD:

- Register 7 contains 0,
- The first operand points to the byte before the leftmost byte in the source field, and
- The second operand points to the byte before the leftmost byte in the destination field.

Indicators

The carry and overflow indicators are unchanged. The remaining indicators reflect the last byte moved.

MVFD Example

```
MVFD (R5),(R6)
```

Assume that:

- Register 5 contains X'0200',
- Register 6 contains X'0300', and
- Register 7 contains X'0003'.

The address of the rightmost byte of the source field is 0200, and 0300 is the address of the rightmost byte of the destination field. As Figure 4-3 shows, MVFD moves 3 bytes:

(1) Byte 0200 to byte 0300,
(2) Byte 01FF to byte 02FF, and
(3) Byte 01FE to byte 02FE.

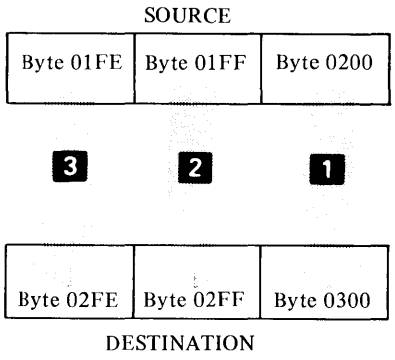


Figure 4-3. MVFD example.

When MVFD is finished:

- Register 7 contains X'0000',
- Register 5 contains X'01FD', and
- Register 6 contains X'02FD'.

Move Byte Field and Increment (MVFN)

This instruction moves a specified number of bytes, left-to-right, from one storage location to another.

Name	*Operation*	*Operand*
[label]	MVFN	(reg), (reg)

MVFN moves a field between two storage locations. For *(reg),(reg)* code the registers that contain the addresses of the leftmost bytes of the source and destination fields.

MVFN assumes that you have loaded register 7 with an unsigned number—the number of bytes to be moved. If R7 contains zero, this instruction is treated as a no-operation.

After MVFN:

- Register 7 contains 0,
- The first operand points to the byte after the rightmost byte in the source field, and
- The second operand points to the byte after the rightmost byte in the destination field.

Indicators

The carry and overflow indicators are unchanged. The remaining indicators reflect the last byte moved.

MVFN Example

```
MVFN (R5),(R6)
```

Assume that:

- Register 5 contains X'0200',
- Register 6 contains X'0300', and
- Register 7 contains X'0003'.

The address of the leftmost byte of the source field is 0200, and 0300 is the address of the leftmost byte of the destination field. As Figure 4-4 shows, MVFN moves 3 bytes:

(1) Byte 0200 to byte 0300,
(2) Byte 0201 to byte 0301, and
(3) Byte 0202 to byte 0302.

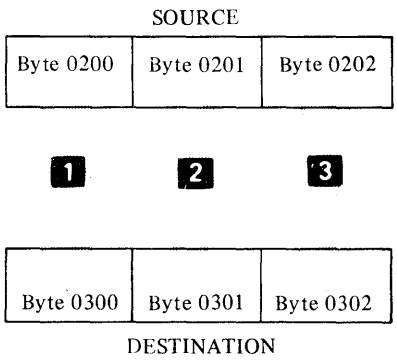


Figure 4-4. MVFN example.

When MVFN is finished:

- Register 7 contains X'0000',
- Register 5 contains X'0203', and
- Register 6 contains X'0303'.

Move Byte Immediate (MVBI)

This instruction places one byte of immediate data into a register.

Name	*Operation*	*Operand*
[label]	MVBI	byte, reg

For the *byte* operand, code an absolute value or expression, −128 to 127 or 0 to 255. MVBI places this value into bits 8–15 of *reg*. The high-order bit of the *byte* value is propagated through bits 0–7 of *reg*.

Indicators

The carry and overflow indicators are unchanged. The remaining indicators reflect the byte loaded into *reg*.

MVBI Example

```
MVBI -3,R6
```

This instruction places −3 into bits 8–15 of register 6. Bits 0–7 contain ones.

Move Doubleword (MVD)

This instruction moves a doubleword (4 bytes):

- From a register pair to storage,
- From storage to a register pair, or
- From storage to storage.

Name	*Operation*	*Operand*
[label]	MVD	reg, addr4 addr4, reg addr5, addr4

For the register-to-storage syntax or the storage-to-register syntax, specify a storage address and the first register of a register pair.

For the storage-to-storage syntax, specify the addresses of the source and destination of the doubleword.

Indicators

The carry and overflow indicators are unchanged. The remaining indicators reflect the doubleword moved into the second operand.

MVD Examples

```
MVD THERE,HERE
```

This instruction moves a doubleword from storage location THERE to storage location HERE.

```
MVD R4,(R6)
```

This instruction moves a doubleword—the contents of registers 4 and 5—to the storage location indicated by the contents of register 6.

```
MVD LOC2,R7
```

This instruction moves a doubleword from storage location LOC2 and places it in registers 7 and 0.

Move Doubleword and Zero (MVDZ)

This instruction moves a doubleword (4 bytes) from storage to a register pair, then replaces the doubleword in storage with zeros.

Name	*Operation*	*Operand*
[label]	MVDZ	addr4, reg

When you code MVDZ, specify a storage address and the first register of a register pair. MVDZ moves a doubleword from *addr4* to the register pair.

After the move, the doubleword at *addr4* is filled with zeros.

Indicators

The carry and overflow indicators are unchanged. The remaining indicators reflect the doubleword moved into the second operand.

MVDZ Example

```
MVDZ (R2),R3
```

This instruction moves a doubleword *from* the address in register 2 *to* registers 3 and 4. After the move, the doubleword whose address is in register 2 is filled with zeros.

Move Word (MVW)

This instruction moves a word (2 bytes):

- From a register to a register,
- From a register to storage,
- From storage to a register, or
- From storage to storage.

Name	*Operation*	*Operand*
[label]	MVW	reg, reg reg, addr4 addr4, reg longaddr, reg reg, longaddr addr5, addr4

Indicators

The carry and overflow indicators are unchanged. The remaining indicators reflect the word moved into the second operand.

MVW Examples

```
MVW R1,R2
```

This instruction (coded in *reg,reg* form) moves a word from register 1 to register 2.

```
MVW (R1),R2
```

This instruction (coded in *addr4,reg* form) moves a word *from* the storage location whose address is in register 1 *to* register 2.

Note. This instruction is also a valid *longaddr,reg* form. The assembler generates the *addr4,reg* form because it is more efficient with respect to speed and storage usage.

```
MVW (R1),(R2)
```

This instruction (coded in *addr5,addr4* form) moves a word *from* the storage location whose address is in register 1 *to* the storage location whose address is in register 2.

Move Word and Zero (MVWZ)

This instruction moves a word (2 bytes) from storage to a register, then fills the word in storage with zeros.

Name	*Operation*	*Operand*
[label]	MVWZ	addr4, reg

Indicators

The carry and overflow indicators are unchanged. The remaining indicators reflect the word moved into the second operand.

MVWZ Example

```
MVWZ (R2,DISP5),R6
```

The address in register 2, plus the value of DISP5, form the address of the word to be moved. The word is moved to register 6. After the move, the word in storage is filled with zeros.

Move Word Immediate (MVWI)

This instruction moves a one-word (2-byte) absolute value to a storage location or into a register.

Name	*Operation*	*Operand*
[label]	MVWI	word, addr4 word, reg

For the *word* operand, code an absolute value or expression in the range of −32768 to +32767 or 0 to 65535.

Indicators

The carry and overflow indicators are unchanged. The remaining indicators reflect the word moved into the second operand.

MVWI Example

```
MVWI 3488,LOC3
```

This instruction moves the value 3488 into the word at storage location LOC3.

Move Word Short (MVWS)

This instruction moves a word (2 bytes):

- From a register to storage, or
- From storage to a register.

Name	*Operation*	*Operand*
[label]	MVWS	reg, shortaddr shortaddr, reg

The operand shortaddr is an address value that you code in one of the following forms:

(reg$^{0\text{-}3}$,wdisp)
The effective address is the value of *wdisp* added to the contents or *reg*$^{0\text{-}3}$.

*(reg$^{0\text{-}3}$,wdisp)**
The effective address is the contents of storage at the address defined by the value of *wdisp* added to the contents of *reg*$^{0\text{-}3}$.

(reg$^{0\text{-}3}$)
The effective address is the contents of (*reg*$^{0\text{-}3}$).

*(reg$^{0\text{-}3}$)**
The effective address is the contents of storage at the address defined by the contents of *reg*$^{0\text{-}3}$.

addr
To use this form, the instruction is in the domain of a USING directive and *addr* is in the range of the same USING directive. The assembler computes a displacement (0–62) and register combination that references the requested location.

*addr**
Same as *addr*, except the assembler computes the effective address as the contents of storage at the address defined by a displacement (0–62) and register combination.

Note. For *addr* and *addr**, the base register must be *reg*$^{0\text{-}3}$.

Indicators

The carry and overflow indicators are unchanged. The remaining indicators reflect the word moved into the second operand.

MVWS Examples

```
MVWS R6,(R2)
```

This instruction moves the contents of register 6 into the storage location whose address is in register 2.

```
MVWS (R3,24)*,R5
```

- The contents of register 3, plus 24, form an address.
- That storage location contains the address of the word to be moved into register 5.
- MVWS moves this word into register 5.

Coding Hints

Use this instruction to move to or from an address that is either:

- The contents of a register with no displacement, or
- The contents of a register plus a displacement of 0 to 62.

The advantage of using MVWS is that it requires only 2 bytes of storage.

Arithmetic Instructions

Add Address (AA)

This instruction adds an address value to either a register or a word in storage.

Name	*Operation*	*Operand*
[label]	AA	raddr, reg [,reg] raddr, addr4

In the first format, note the optional third operand. If you code this register, AA places its result there, leaving the first and second operands unchanged. Otherwise, AA places its result in the second operand, leaving the first operand unchanged.

Indicators

The overflow indicator is cleared. If the addition results in a sum that is less than -2^{15} or greater than $+2^{15}-1$, the overflow indicator is turned on.

The carry indicator is turned on if the addition results in a carry out of the high-order bit position of the result operand (for a total of 17 bits in the sum). If there is no carry, the carry indicator is turned off.

Also, if an overflow occurs, the carry indicator contains the sign bit, and the result operand contains the low-order 16 bits of the sum.

The other indicators change to reflect the 16-bit result.

AA Example

```
AA   DATA,R0,R3
```

In this example, AA adds the address of DATA to the contents of R0, and places the result in R3.

Add Byte (AB)

This instruction adds:

- A byte in a register to a byte in storage, or
- A byte in storage to a byte in a register.

Name	*Operation*	*Operand*
[label]	AB	reg, addr4 addr4, reg

If you code the *reg,addr4* form, bits 8–15 of *reg* are added to the byte at *addr5*.

In the *addr4,reg* form, the byte at *addr4* is added to bits 8–15 of *reg*. The high-order byte of *reg* remains unchanged.

Indicators

The overflow indicator is cleared. If the addition results in a sum that is less than -2^{7} or greater than $+2^{7}-1$, the overflow indicator is turned on.

The carry indicator is turned on if the addition results in a carry out of the high-order bit position of the byte (for a total of 9 bits in the sum). If there is no carry, the carry indicator is turned off.

Also, if an overflow occurs, the carry indicator contains the sign bit, and the second operand contains the low-order 8 bits of the sum.

The other indicators change to reflect the 8-bit result.

AB Examples

```
AB VALØ1,R5
```

Assume that VAL01 contains X'20', and register 5 contains X'2C83'. AB adds: 20+83=A3. VAL01 remains unchanged, and register 5 now contains X'2CA3'. The carry indicator and overflow indicator are both off.

```
AB VALØ2,R5
```

Assume that VAL02 contains X'30', and register 5 contains X'08FE'. AB adds: 30+FE=12E. VAL02 remains unchanged, and register 5 now contains X'082E'. The carry indicator is on and the overflow indicator is off.

Add Byte Immediate (ABI)

This instruction adds a 1-byte absolute value to a register.

Name	*Operation*	*Operand*
[label]	ABI	byte, reg

For *byte*, code an 8-bit value in the range of −128 to +127 or 0 to 255. ABI expands this value to 16 bits by propagating the sign bit to the left of the *byte* value. This value is then added to *reg*.

Indicators

The overflow indicator is cleared. If the addition results in a sum that is less than -2^{15} or greater than $+2^{15}-1$, the overflow indicator is turned on.

The carry indicator is turned on if the addition results in a carry out of the high-order bit position of the register (for a total of 17 bits in the sum). If there is no carry, the carry indicator is turned off.

Also, if an overflow occurs, the carry indicator contains the sign bit, and the register contains the low-order 16 bits of the sum.

The other indicators change to reflect the 16-bit result.

ABI Example

```
ABI 34,R6
```

Assume that R6 contains X'0050'. ABI expands X'22' (the equivalent of decimal 34) to 16 bits by propagating the sign bit (zero) to the left. ABI then adds: X'0022'+X'0050'=X'0072'. R6 now contains X'0072' (decimal 114).

Add Carry Indicator (ACY)

This instruction adds the value of the carry indicator to a register.

Name	*Operation*	*Operand*
[label]	ACY	reg

Indicators

The overflow indicator is cleared. If the addition results in a sum that is less than -2^{15} or greater than $+2^{15}-1$, the overflow indicator is turned on.

The carry indicator is turned on if the addition results in a carry out of the high-order bit position of the register (for a total of 17 bits in the sum). If there is no carry, the carry indicator is turned off.

Also, if an overflow occurs, the carry indicator contains the sign bit, and the register contains the low-order 16 bits of the sum.

If the zero indicator is on at the beginning of this instruction, it is set to reflect the result; if it is off at the beginning, it stays off. The negative indicator reflects the sum, and the even indicator is unchanged.

ACY Example

```
ACY R4
```

Assume that register 4 contains X'0027', and the carry indicator is on. ACY adds 1 to the contents of R4, and the register now contains X'0028'.

Add Doubleword (AD)

This instruction adds:

- A doubleword (4 bytes) in a register pair to a doubleword in storage,
- A doubleword in storage to a doubleword in a register pair, or
- A doubleword in storage to a doubleword in storage.

Name	*Operation*	*Operand*
[label]	AD	reg, addr4 addr4, reg addr5, addr4

In either the register-to-storage form or the storage-to-register form, code—for the *reg* operand—the first register of a register pair. For example, if you code R5, AD uses registers 5 and 6. If you code R7, AD uses registers 7 and 0.

AD adds the contents of the doubleword specified by the first operand to the contents of the doubleword specified by the second operand, placing its result in the second operand. The first operand remains unchanged.

Indicators

The overflow indicator is cleared. If the sum is less than -2^{31} or greater than $+2^{31}-1$, the overflow indicator is turned on.

The carry indicator is turned on if the addition results in a carry out of the high-order bit position of the sum (for a total of 33 bits in the sum). If there is no carry, the carry indicator is turned off.

Also, if an overflow occurs, the carry indicator contains the sign bit, and the register pair or doubleword in storage contains the low-order 32 bits of the sum.

The other indicators change to reflect the 32-bit result.

AD Example

```
AD R3,(R1)
```

Assume that the register pair R3,R4 contains X'25000000', R1 contains X'0300', and the doubleword at storage address 0300 contains X'10000000'. AD adds: X'25000000'+X'10000000'=X'35000000'. Registers 3 and 4 remain unchanged, and the doubleword at storage address 300 contains X'35000000'.

Add Word (AW)

This instruction adds:

- A word (2 bytes) in a register to a word in a register,
- A word in a register to a word in storage,
- A word in storage to a word in a register, or
- A word in storage to a word in storage.

Name	*Operation*	*Operand*
[label]	AW	reg, reg reg, addr4 addr4, reg longaddr, reg addr5, addr4

AW adds the contents of the word specified by the first operand to the contents of the word specified by the second operand. The first operand remains unchanged.

Indicators

The overflow indicator is cleared. If the addition results in a sum that is less than -2^{15} or greater than $+2^{15}-1$, the overflow indicator is turned on.

The carry indicator is turned on if the addition results in a carry out of the high-order bit position of the result operand (for a total of 17 bits in the sum). If there is no carry, the carry indicator is turned off.

Also, if an overflow occurs, the carry indicator contains the sign bit, and the result operand contains the low-order 16 bits of the sum.

The other indicators change to reflect the 16-bit result.

AW Example

```
AW THERE,(R2)
```

This instruction adds the word at storage location THERE to the word at the storage location whose address is in R2.

Add Word Immediate (AWI)

This instruction adds a 1-word (2-byte) absolute value:

- To a register, or
- To the contents of a storage location.

Name	*Operation*	*Operand*
[label]	AWI	word, reg[,reg] word, addr4

For *word*, code a 16-bit value in the range −32768 to +32767 or 0 to 65535. AWI adds this value to the contents of the word specified by the second operand.

In the *word,reg*[,
register for this operand, the result of the addition is placed in that register. If you do not code the third operand, AWI places the sum in the register specified by the second operand.

Indicators

The overflow indicator is cleared. If the addition results in a sum that is less than -2^{15} or greater than $+2^{15}-1$, the overflow indicator is turned on.

The carry indicator is turned on if the addition results in a carry out of the high-order bit position of the result operand (for a total of 17 bits in the sum). If there is no carry, the carry indicator is turned off.

Also, if an overflow occurs, the carry indicator contains the sign bit, and the result operand contains the low-order 16 bits of the sum.

The other indicators change to reflect the 16-bit result.

AWI Example

```
AWI 25Ø2,R3,R1
```

In this example, AWI adds the decimal value 2502 to the contents of R3, and places the result in R1. R3 is unchanged.

Add Word with Carry (AWCY)

This instruction adds the contents of a specified register, plus the value of the carry indicator, to another register.

Name *	*Operation*	*Operand*
[label]	AWCY	reg, reg

AWCY places the final sum of the register specified by the first operand, the register specified by the second operand, and the carry indicator in the register specified by the second operand. The contents of the first register are unchanged.

Indicators

The overflow indicator is cleared. If the addition results in a sum that is less than -2^{15} or greater than $+2^{15}-1$, the overflow indicator is turned on.

The carry indicator is turned on if the addition results in a carry out of the high-order bit position of the word (for a total of 17 bits in the sum). If there is no carry, the carry indicator is turned off.

Also, if an overflow occurs, the carry indicator contains the sign bit, and the second operand contains the low-order 16 bits of the sum.

If the zero indicator is on at the beginning of this instruction, it is set to reflect the result; if it is off at the beginning, it stays off. The negative indicator reflects the sum, and the even indicator is unchanged.

AWCY Example

```
AWCY R6,R4
```

Assume that the instruction just before this AWCY left the carry indicator on. This instruction adds the contents of R6, plus 1, to the contents of R4. Register 4 contains the result, and register 6 remains unchanged.

Subtract Address (SA)

This instruction subtracts an address value from either a word in a register or a word in storage.

Name	*Operation*	*Operand*
[label]	SA	raddr, reg [,reg] raddr, addr4

In the first format, note the optional third operand. If you code this register, SA places its result there, leaving the first and second operands unchanged. Otherwise, SA places its result in the second operand, leaving the first operand unchanged.

Indicators

The overflow indicator is cleared. If the subtraction results in a difference that is less than -2^{15} or greater than $+2^{15}-1$, the overflow indicator is turned on.

The carry indicator is turned on if the subtraction results in a borrow into the high-order bit position of the result operand. If there is no borrow, the carry indicator is turned off.

Also, if an overflow occurs, the carry indicator contains the complement of the sign bit, and the result operand contains the low-order 16 bits of the difference.

The other indicators change to reflect the 16-bit result.

SA Example

```
SA   DATA2,(R2)
```

In this example, R2 contains the address of a word in storage. SA subtracts the address of DATA2 from that word, replacing the word in storage with the result. DATA2 and R2 are unchanged.

Subtract Byte (SB)

This instruction subtracts either:

- A byte in a register from a byte in storage, or
- A byte in storage from a byte in a register.

Name	*Operation*	*Operand*
[label]	SB	reg, addr4 addr4, reg

If you code the *reg,addr4* form, bits 8–15 of *reg* are subtracted from the byte at *addr4*.

In the *addr4,reg* form, the byte at *addr4* is subtracted from bits 8–15 of *reg*. The high-order byte of *reg* remains unchanged.

Indicators

The overflow indicator is cleared. If the subtraction results in a difference that is less than -2^{7} or greater than $+2^{7}-1$, the overflow indicator is turned on.

The carry indicator is turned on if the subtraction results in a borrow beyond the high-order bit position of the byte. If there is no borrow, the carry indicator is turned off.

Also, if an overflow occurs, the carry indicator contains the complement of the sign bit, and the second operand contains the low-order 8 bits of the difference.

The other indicators change to reflect the 8-bit result.

SB Example

```
SB VALØ1,R5
```

In this example, assume that VAL01 contains X'20', and R5 contains X'2C83'. SB subtracts: X'83'−X'20'=X'63'. VAL01 remains unchanged, and register 5 now contains X'2C63'. The carry and overflow indicators are off.

Subtract Carry Indicator (SCY)

This instruction subtracts the value of the carry indicator from a register.

Name	*Operation*	*Operand*
[label]	SCY	reg

Indicators

The overflow indicator is cleared. If the subtraction results in a difference that is less than -2^{15} or greater than $+2^{15}-1$, the overflow indicator is turned on.

The carry indicator is turned on if the subtraction results in a borrow beyond the high-order bit position of the register. If there is no borrow, the carry indicator is turned off.

Also, if an overflow occurs, the carry indicator contains the complement of the sign bit, and the register contains the low-order 16 bits of the difference.

If the zero indicator is on at the beginning of this instruction, it is set to reflect the result; if it is off at the beginning, it stays off. The negative indicator reflects the sum, and the even indicator is unchanged.

SCY Example

```
SCY R4
```

Assume that R4 contains X'0027', and the carry indicator is on. SCY subtracts 1 from the contents of R4, and the register now contains X'0026'.

Subtract Doubleword (SD)

This instruction subtracts:

- The contents of a register pair from a doubleword (4 bytes) in storage,
- A doubleword in storage from the contents of a register pair, or
- A doubleword in storage from another doubleword in storage.

Name	*Operation*	*Operand*
[label]	SD	reg, addr4 addr4, reg addr5, addr4

In either the register-to-storage form or the storage-to-register form, code—for *reg*—the first register of a pair. For example, if you code R2, SD uses registers 2 and 3. If you code R7, SD uses registers 7 and 0.

SD subtracts the contents of the doubleword specified by the first operand *from* the contents of the doubleword specified by the second operand. The first operand remains unchanged.

Indicators

The overflow indicator is cleared. If the subtraction results in a difference that is less than -2^{31} or greater than $+2^{31}-1$, the overflow indicator is turned on.

The carry indicator is turned on if the subtraction results in a borrow beyond the high-order bit position. If there is no borrow, the carry indicator is turned off.

Also, if an overflow occurs, the carry indicator contains the complement of the sign bit, and the second operand contains the low-order 32 bits of the difference.

The other indicators change to reflect the 32-bit result.

SD Example

```
SD R3,(R1)
```

In this example, assume that registers 3 and 4 contain X'10000000', and register 1 contains X'0300'. The doubleword at storage address 0300 contains X'25000000'. SD subtracts: X'25000000'−X'10000000'=X'15000000'. Registers 3 and 4 remain unchanged, and the doubleword at storage address 0300 contains X'15000000'.

Subtract Word (SW)

This instruction subtracts:

- A register from a register,
- A register from a word (2 bytes) in storage,
- A word in storage from a register, or
- A word in storage from a word in storage.

Name	*Operation*	*Operand*
[label]	SW	reg, reg reg, addr4 addr4, reg longaddr, reg addr5, addr4

SW subtracts the contents of the word specified by the first operand *from* the contents of the word specified by the second operand. SW places its result in the second operand.

Indicators

The overflow indicator is cleared. If the subtraction results in a difference that is less than -2^{15} or greater than $+2^{15}-1$, the overflow indicator is turned on.

The carry indicator is turned on if the subtraction results in a borrow beyond the high-order bit position of the result operand. If there is no borrow, the carry indicator is turned off.

Also, if an overflow occurs, the carry indicator contains the complement of the sign bit, and the result operand contains the low-order 16 bits of the difference.

The other indicators change to reflect the 16-bit result.

SW Example

```
SW THERE,(R2)
```

This instruction subtracts the word at location THERE from the word in storage whose address is in R2.

Subtract Word Immediate (SWI)

This instruction subtracts a 1-word (2-byte) absolute value from a register *or* from the contents of a storage location.

Name	*Operation*	*Operand*
[label]	SWI	word, reg[,reg] word, addr4

For *word*, code a 16-bit value in the range −32768 to +32767 or 0 to 65535. SWI subtracts this value from the contents of the word specified by the second operand.

In the *word,reg[reg]* format there is an optional third operand. If you code a register for this operand, the result of the subtraction is placed in that register. If you do not code the third operand, SWI places the difference in the register specified by the second operand.

Indicators

The overflow indicator is cleared. If the subtraction results in a difference that is less than -2^{15} or greater than $+2^{15}-1$, the overflow indicator is turned on.

The carry indicator is turned on if the subtraction results in a borrow beyond the high-order bit position of the result operand. If there is no borrow, the carry indicator is turned off.

Also, if an overflow occurs, the carry indicator contains the complement of the sign bit, and the result operand contains the low-order 16 bits of the difference.

The other indicators change to reflect the 16-bit result.

SWI Example

```
SWI 25Ø2,R3,R1
```

In this example, SWI subtracts the decimal value 2502 from the contents of R3, and places the result in R1.

Subtract Word with Carry (SWCY)

This instruction subtracts the contents of one register and the carry indicator from the contents of another register.

Name	*Operation*	*Operand*
[label]	SWCY	reg, reg

SWCY subtracts the contents of the first register and the carry indicator from the contents of the second register. SWCY places the final result in the second register, leaving the first register unchanged.

Indicators

The overflow indicator is cleared. If the subtraction results in a difference that is less than -2^{15} or greater than $+2^{15}-1$, the overflow indicator is turned on.

The carry indicator is turned on if the subtraction results in a borrow beyond the high-order bit position of the register. If there is no borrow, the carry indicator is turned off.

Also, if an overflow occurs, the carry indicator contains the complement of the sign bit, and the second register contains the low-order 16 bits of the difference.

If the zero indicator is on at the beginning of this instruction, it is set to reflect the result; if it is off at the beginning, it stays off. The negative indicator reflects the difference, and the even indicator is unchanged.

SWCY Example

```
SWCY R6,R4
```

Assume that the instruction just before this SWCY left the carry indicator on. This instruction subtracts the contents of R6 from R4, then decreases the difference by 1. R4 contains the result, and R6 remains unchanged.

Multiply Byte (MB)

This instruction multiplies the contents of a register by a byte in storage.

Name	*Operation*	*Operand*
[label]	MB	addr4, reg

MB multiplies the contents of *reg* by the byte at *addr4*. The result (1 word) is placed in *reg*.

Indicators

The carry and overflow indicators are cleared. If the product of the multiplication cannot be represented in 16 bits, the overflow indicator is turned on. If there is an overflow, the contents of the result register are undefined. The remaining indicators change to reflect the result.

MB Example

```
MB (R3,25),R6
```

In this example, assume that R3 contains X'0400' and register 6 contains X'0035'.

MB determines that the address of the byte to be multiplied is X'0419' (25 bytes past the address in R3). Assume that this byte contains X'11'.

MB multiplies: X'11'X'0035'=X'0385'. This result is placed in register 6.

Multiply Doubleword (MD)

This instruction multiplies a word in storage by the contents of a register pair.

Name	*Operation*	*Operand*
[label]	MD	addr4, reg

For the *reg* operand, code the first register of a register pair. For example, if you code R1, MD uses registers 1 and 2. If you code R7, MD uses registers 7 and 0.

MD multiplies the word at *addr4* by the contents of the register pair specified by *reg*. The result (1 doubleword) is placed in the register pair.

Indicators

The carry and overflow indicators are cleared. If the product of the multiplication cannot be represented in 32 bits, the overflow indicator is turned on. If there is an overflow, the contents of the register pair are undefined. The remaining indicators change to reflect the result.

MD Example

```
MD 8(R1)*,R7
```

Here is how MD calculates the address of the word to be multiplied:

1. R1 contains the address of a location in storage.
2. The contents of that location are another address.
3. *That* address value is increased by 8 to form the address of the word to be multiplied.

MD multiplies the contents of registers 7 and 0 by this word, and places the result in registers 7 and 0.

Multiply Word (MW)

This instruction multiplies the contents of a register by a word in storage.

Name	*Operation*	*Operand*
[label]	MW	addr4, reg

MW multiplies the contents of *reg* by the word at *addr4*. The result (1 word) is placed in *reg*.

Indicators

The carry and overflow indicators are cleared. If the product of the multiplication cannot be represented in 16 bits, the overflow indicator is turned on. If there is an overflow, the contents of the result register are undefined. The remaining indicators change to reflect the result.

MW Example

```
MW LOC8*,R6
```

The contents of storage at address LOC8 are the address of the word to be multiplied. MW fetches the word and multiplies the contents of R6 by it. The product is in R6.

Divide Byte (DB)

This instruction divides a byte in storage into the contents of a register.

Name	*Operation*	*Operand*
[label]	DB	addr4, reg

DB divides the byte at *addr4* into the contents of *reg*. The quotient is placed in *reg*, and the remainder is placed in the register following the one you coded. For example, if you coded R3, the quotient appears in register 3, and the remainder in register 4. If you coded R7, the quotient appears in register 7, and the remainder in register 0.

Indicators

The overflow indicator is cleared. If you tried to divide by zero, or if the quotient cannot be represented in 16 bits, the overflow indicator is turned on. If there is an overflow, the result of the division and the remaining indicators are undefined.

If you tried to divide by zero, the carry indicator is also turned on; otherwise, the carry indicator is cleared. The other indicators change to reflect the quotient.

DB Example

```
DB (R1),R6
```

DB divides the byte whose address is in R1 *into* the contents of R6. The quotient is in R6 and the remainder is in R7.

Divide Doubleword (DD)

This instruction divides a word in storage into the contents of a register pair.

Name	*Operation*	*Operand*
[label]	DD	addr4, reg

For the *reg* operand, code the first register of a register pair. For example, if you code R2, DD uses registers 2 and 3. If you code R7, DD uses registers 7 and 0.

DD divides the word at *addr4* into the contents of the register pair specified by *reg*. The quotient is placed in the register pair, and the remainder is placed in the register following the second register of the pair. For example, if you coded R3, the quotient is placed in registers 3 and 4, and the remainder in register 5.

Indicators

The overflow indicator is cleared. If you tried to divide by zero, or if the quotient cannot be represented in 32 bits, the overflow indicator is turned on. If there is an overflow, the result of the division and the remaining indicators are undefined.

If you tried to divide by zero, the carry indicator is also turned on; otherwise, the carry indicator is cleared. The other indicators change to reflect the quotient.

DD Example

```
DD (R1)*,R6
```

In this example, the storage location whose address is in R1 contains the *address* of the word to be used. DD divides this word into the doubleword in registers 6 and 7. The quotient is in registers 6 and 7, and the remainder is in register 0.

Divide Word (DW)

This instruction divides a word in storage into the contents of a register.

Name	*Operation*	*Operand*
[label]	DW	addr4, reg

DW divides the word at *addr4* into the contents of *reg*. The quotient is placed in *reg*, and the remainder is placed in the register following the one you coded. For example, if you coded R5, the quotient appears in register 5, and the remainder in register 6. If you coded R7, the quotient appears in register 7, and the remainder in register 0.

Indicators

The overflow indicator is cleared. If you tried to divide by zero, or if the quotient cannot be represented in 16 bits, the overflow indicator is turned on. If there is an overflow, the result of the division and the remaining indicators are undefined.

If you tried to divide by zero, the carry indicator is also turned on; otherwise, the carry indicator is cleared. The other indicators change to reflect the quotient.

DW Example

```
DW WORD5+4,R7
```

In this example, the word to be used is 4 bytes past WORD5. DW divides this word into the contents of R7. R7 contains the quotient, and R0 contains the remainder.

Complement Register (CMR)

This instruction places the complement (in twos complement form) of the contents of a register back into the same register or, optionally, into another register.

Name	*Operation*	*Operand*
[label]	CMR	reg[,reg]

Note the optional second operand. If you code this register, CMR places the complement of the first register into the second, leaving the first operand unchanged. Otherwise, CMR places the complement back into the source register.

Indicators

The overflow indicator is cleared. If the number to be complemented is −32768 (X'8000'), the overflow indicator is turned on.

The carry indicator is cleared. If the number to be complimented is zero, the carry indicator is turned on. The remaining indicators change to reflect the result.

CMR Examples

```
CMR RØ
```

Assume that R0 contains X'0003'. CMR places its complement, X'FFFD', into R0.

```
CMR RØ,R6
```

Assume that R0 contains X'0003'. CMR places its complement, X'FFFD', into R6, leaving R0 unchanged.

Branching Instructions

Branch (B)

This instruction causes an unconditional branch to the address specified by *longaddr*.

Name	*Operation*	*Operand*
[label]	B	longaddr

Indicators

All indicators are unchanged.

B Example

```
B (R6,LOC1+4)*
```

This instruction branches to the location whose address it calculates as follows:

1. Register 6 contains an address.
2. The contents of R6, plus the value of LOC1, plus 4, form an address.
3. The contents of *that* storage location specify the branch address.

Branch and Link (BAL)

This instruction saves—in a register—the address of the next sequential instruction, then branches to *longaddr*.

Name	*Operation*	*Operand*
[label]	BAL	longaddr, reg

Note. If the same register specified as the second operand is also used as a base register in *longaddr*, the initial contents of that register are first used in effective address computation and then overwritten with the address of the next sequential instruction.

Indicators

All indicators are unchanged.

BAL Example

```
BAL NEXT,R7
```

In this example, BAL:

- Determines the address of NEXT,
- Saves (in register 7) the address of the next sequential instruction, and then
- Branches to NEXT.

Branch and Link External (BALX)

This instruction causes an unconditional branch to an address in another source module. It saves—in a register—the address of the instruction that follows the BALX instruction.

Name	*Operation*	*Operand*
[label]	BALX	vcon, reg

For *vcon*, code the external symbol that defines the location to be branched to. For *reg*, code the register that you want to load with the address of the next sequential instruction.

Note. You need not code an EXTRN statement to define the external symbol specified by *vcon*. The *vcon* symbol must be a valid entry point in another source module. The application builder will resolve the reference between modules.

Indicators

All indicators are unchanged.

BALX Example

```
* SOURCE MODULE A
         :
         :
         BALX  ENTER1,R3
         MVW   VALØ3,R1
         :
         :
*SOURCE MODULE B
         ENTRY ENTER1
         :
ENTER1   EQU   *
         :
         BXS   (R3)
```

Branch and Link Short (BALS)

This instruction saves—in register 7—the address of the next sequential instruction, then branches to the specified address.

Name	*Operation*	*Operand*
[label]	BALS	(reg, jdisp)* (reg)* addr*

Code the address of the location whose *contents* specify the address to be branched to. If you specify the *(reg,jdisp)** form, *jdisp* must be in the range –256 to 254. The *addr** form can be used only when BALS and the address to be branched to are within the domain and range of the same USING statement.

If the implied register (register 7) is used as *reg* in either *(reg,jdisp)** or *(reg)**, the initial contents of register 7 are first used in effective address computation and then overwritten with the address of the next sequential instruction.

Note. BALS is a 2-byte instruction, and uses only indirect addressing.

Indicators

All indicators are unchanged.

BALS Example

```
BALS (R3,28)*
```

In this example, BALS:

- calculates an address as follows:
  (1) R3 contains an address.
  (2) This address is increased by 28.
  (3) The result is the address of the location that contains the branch address.
- then saves (in register 7) the address of the next sequential instruction, and
- branches to the address calculated.

Branch External (BX)

This instruction causes an unconditional branch to an address in another source module.

Name	*Operation*	*Operand*
[label]	BX	vcon

For *vcon*, code the external symbol that defines the branch location.

Note. The *vcon* symbol must be a valid entry point in another source module. You need not code an EXTRN statement to define the external symbol specified by *vcon*. The application builder will resolve the reference between modules.

Indicators

All indicators are unchanged.

BX Example

```
* SOURCE MODULE A
         ENTRY  ENTER2
         :
         :
         BX     ENTER1
ENTER2   EQU    *
         :
         :
* SOURCE MODULE B
         ENTRY  ENTER1
         :
ENTER1   EQU    *
         :
         BX     ENTER2
```

Branch if Mixed (BMIX)

After a Test Word Immediate (TWI) instruction, BMIX causes a branch if the bits tested by TWI are a combination of zeros and ones.

Note. BMIX actually tests the zero and negative indicators.

Name	*Operation*	*Operand*
[label]	BMIX	longaddr

Indicators

All indicators are unchanged.

BMIX Example

Assume that R4 contains X'002B'.

```
TWI   X'ØØ36',R4
BMIX  (R2)
```

Because the bits tested by TWI are a combination of zeros and ones, BMIX causes a branch to the address in R2.

Branch if Not Mixed (BNMIX)

After a Test Word Immediate (TWI) instruction, BNMIX causes a branch if the bits tested by TWI are either all zeros or all ones.

Note. BNMIX actually tests the zero and negative indicators.

Name	*Operation*	*Operand*
[label]	BNMIX	longaddr

Note. If the first operand (the 1-word mask) of the TWI instruction is all zeros, the resulting condition is *not mixed.* In this case, BNMIX causes a branch.

Indicators

All indicators are unchanged.

BNMIX Example

Assume that the word whose address is in R1 contains X'000F'.

```
TWI   X'ØØFØ',(R1)
BNMIX (R6,4)
```

Because the bits tested by TWI are all zeros, BNMIX causes a branch to the location that is 4 bytes past the address in R6.

Branch if Not Off (BNOFF)

After a Test Bit (TBT) or Test Word Immediate (TWI) instruction, BNOFF causes a branch if:

- The bit tested by TBT is on, or
- The bits tested by TWI are either mixed or all on.

Name	*Operation*	*Operand*
[label]	BNOFF	longaddr

Note. BNOFF actually tests the zero indicator.

Indicators

All indicators are unchanged.

BNOFF Example

Assume that the word at location TEST contains X'0246'.

```
TWI   X'Ø369',TEST
BNOFF (R3)
```

Because the bits tested by TWI are mixed, BNOFF causes a branch to the address in R3.

Branch if Not On (BNON)

After a Test Bit (TBT) or Test Word Immediate (TWI) instruction, BNON causes a branch if:

- The bit tested by TBT is off, or
- The bits tested by TWI are either mixed or all off.

Note. BNON actually tests the negative indicator.

Name	*Operation*	*Operand*
[label]	BNON	longaddr

Note. If the first operand (the 1-word mask) of a TWI instruction is all zeros, the resulting condition is *not on*. In this case, BNON causes a branch.

Indicators

All indicators are unchanged.

BNON Example

Assume that the word whose address is in R3 contains X'00FF'.

```
TWI   X'ØØF1',(R3)
BNON  LOC4
```

Because the bits tested by TWI are all on, BNON does *not* cause a branch to LOC4.

Branch if Off (BOFF)

After a Test Bit (TBT) or Test Word Immediate (TWI) instruction, BOFF causes a branch if:

- The bit tested by TBT is off, or
- The bits tested by TWI are all off.

Note. BOFF actually tests the zero indicator.

Name	*Operation*	*Operand*
[label]	BOFF	longaddr

Note. If the first operand (the 1-word mask) of a TWI instruction is all zeros, the resulting condition is *off*. In this case BOFF causes a branch.

Indicators

All indicators are unchanged.

BOFF Example

```
TBT   (RØ,7)
BOFF  OFF+2
```

Assume that the byte whose address is in R0 contains:
0110 1001
Because the eighth bit is on, BOFF does *not* cause a branch to the address that is 2 bytes past location OFF.

Branch if On (BON)

After a Test Bit (TBT) or a Test Word Immediate (TWI) instruction, BON causes a branch if:

- The bit tested by TBT is on, or
- The bits tested by TWI are all on.

Name	*Operation*	*Operand*
[label]	BON	longaddr

Note. BON actually tests the negative indicator.

Indicators

All indicators are unchanged.

BON Example

```
TBT (R2,3)
BON (R4)*
```

Assume that the byte whose address is in R2 contains:
0111 0011
Because the fourth bit is on, BON causes a branch to the address defined by the contents of the location whose address is in R4.

Branch Indexed Short (BXS)

This instruction causes an unconditional branch to the specified address.

Name	*Operation*	*Operand*
[label]	BXS	(reg $^{1-7}$, jdisp) (reg $^{1-7}$) addr

In the (*reg* $^{1-7}$,
form can be used only when BXS and the address to be branched to are within the domain and range of the same USING statement.

Note. BXS is a 2-byte instruction.

Indicators

All indicators are unchanged.

BXS Example

```
BXS (R2,2)
```

In this example, BXS causes a branch to the location that is 2 bytes past the address in register 2.

Branch on Carry (BCY)

This instruction tests the carry indicator. If the indicator is on, BCY branches to *longaddr*. If the indicator is off, the branch is not taken, and the next sequential instruction is executed.

Name	*Operation*	*Operand*
[label]	BCY	longaddr

Indicators

All indicators are unchanged.

BCY Example

```
BCY THERE+6
```

In this example, assume that BCY found the carry indicator on. BCY branches to the location that is 6 bytes past THERE.

Branch on Condition (BC)

This instruction tests a condition that you specify. If the tested condition is met, BC causes a branch to *longaddr*. If the condition is not met, the branch is not taken, and the next sequential instruction is executed.

Name	*Operation*	*Operand*
[label]	BC	cond, longaddr

Code BC to test the indicator settings that result from a previous instruction. For the *cond* operand, code the value of the condition you want to test:

Condition Value	*Condition*
0	Zero or equal
1	Positive and non-zero
2	Negative
3	Even
4	Arithmetically less than
5	Arithmetically less than or equal
6	Logically less than or equal
7	Logically less than (carry)

Indicators

All indicators are unchanged.

BC Example

```
BC 2,NEG3
```

In this example, assume that a previous instruction set the negative result indicator on. BC causes a branch to NEG3.

Branch on Condition Code (BCC)

This instruction tests the even, carry, and overflow indicators. If the tested condition code is met, BCC branches to *longaddr*. If the condition code is not met, the branch is not taken, and the next sequential instruction is executed.

Name	*Operation*	*Operand*
[label]	BCC	cond, longaddr

Code BCC to test the indicator settings that result from a previous instruction. For the *cond* operand, specify the condition code you want to test:

Condition Code	*Indicators*
0	E = 0, C = 0, V = 0
1	E = 0, C = 0, V = 1
2	E = 0, C = 1, V = 0
3	E = 0, C = 1, V = 1
4	E = 1, C = 0, V = 0
5	E = 1, C = 0, V = 1
6	E = 1, C = 1, V = 0
7	E = 1, C = 1, V = 1

Abbreviations used for the indicators are:

- E—even
- C—carry
- V—overflow.

Indicators

All indicators are unchanged.

BCC Example

```
BCC 5,(R3)
```

In this example, assume that a previous instruction left the even indicator on, and the carry and overflow indicators off. Because only the even indicator is on, BCC does *not* cause a branch to the address in register 3.

Branch on Equal (BE)

This instruction tests the indicator settings that result from a previous instruction, such as a compare, for an *equal* condition. If the condition is met, BE causes a branch to *longaddr*. If the condition is not met, the branch is not taken, and the next sequential instruction is executed.

Name	*Operation*	*Operand*
[label]	BE	longaddr

Note. This instruction actually tests the zero result indicator.

Indicators

All indicators are unchanged.

BE Example

```
BE EQUAL
```

Assume that this BE was preceded by a compare instruction whose result was *equal*. BE causes a branch to EQUAL.

Branch on Error (BER)

This instruction tests the condition code (after an I/O operation) for an error condition. If there is an error, BER causes a branch to *longaddr*.

Name	*Operation*	*Operand*
[label]	BER	longaddr

Note. Coding this instruction does the same thing as coding the BNCC instruction to branch on condition code *not* 7.

Indicators

All indicators are unchanged.

Branch on Even (BEV)

This instruction tests the even indicator. If the previous instruction left it on, BEV causes a branch to *longaddr*. If the indicator is off, the branch is not taken, and the next sequential instruction is executed.

Name	*Operation*	*Operand*
[label]	BEV	longaddr

Indicators

All indicators are unchanged.

BEV Example

```
BEV (R4)
```

In this example assume that the previous instruction left the even indicator on. BEV causes a branch to the address in register 4.

Branch on Greater Than (BGT)

This instruction tests the indicator settings that result from a previous instruction for an *arithmetically greater than* condition. If the condition is met, BGT causes a branch to *longaddr*. If the condition is not met, the branch is not taken, and the next sequential instruction is executed.

Name	*Operation*	*Operand*
[label]	BGT	longaddr

Note. This instruction actually tests the negative, overflow, and zero result indicators.

Indicators

All indicators are unchanged.

BGT Example

```
BGT GREATER+6
```

In this example, assume that the previous instruction left an *arithmetically greater than* condition. BGT causes a branch to the location that is 6 bytes past GREATER.

Branch on Greater Than or Equal (BGE)

This instruction tests the indicator settings that result from previous instruction for an *arithmetically greater than* or an *equal* condition. If either condition is met, BGE causes a branch to *longaddr*. If neither condition is met, the branch is not taken, and the next sequential instruction is executed.

Name	*Operation*	*Operand*
[label]	BGE	longaddr

Note. This instruction actually tests the negative and overflow indicators.

Indicators

All indicators are unchanged.

BGE Example

```
BGE (R3)
```

In this example, assume that the previous instruction left a *less than* condition. BGE does *not* cause a branch to the address in register 3.

Branch on Less Than (BLT)

This instruction tests the indicator settings that result from a previous instruction for an *arithmetically less than* condition. If the condition is met, BLT causes a branch to *longaddr*. If the condition is not met, the branch is not taken, and the next sequential instruction is executed.

Name	*Operation*	*Operand*
[label]	BLT	longaddr

Note. This instruction actually tests the negative and overflow indicators.

Indicators

All indicators are unchanged.

BLT Example

```
BLT LESS+3
```

In this example, assume that the previous instruction left an *arithmetically less than* condition. BLT branches to the location that is 3 bytes past address LESS. Note that LESS+3 must be an even byte address.

Branch on Less Than or Equal (BLE)

This instruction tests the indicator settings that result from a previous instruction for an *arithmetically less than* or an *equal* condition. If either condition is met, BLE causes a branch to the address specified by *longaddr*. If neither condition is met, the branch is not taken, and the next sequential instruction is executed.

Name	*Operation*	*Operand*
[label]	BLE	longaddr

Note. This instruction actually tests the negative, overflow, and zero result indicators.

Indicators

All indicators are unchanged.

BLE Example

```
BLE THERE-4
```

In this example, assume that the previous instruction left an *equal* condition. BLE causes a branch to the location that is 4 bytes before address THERE.

Branch on Logically Greater Than (BLGT)

This instruction tests the indicator settings that result from a previous instruction, such as a compare, for a *logically greater than* condition. If the condition is met, BLGT causes a branch to *longaddr*. If the condition is not met, the branch is not taken, and the next sequential instruction is executed.

Name	*Operation*	*Operand*
[label]	BLGT	longaddr

Note. This instruction actually tests the carry and zero indicators. If both are off, the branch is taken. For more information about how the indicators are set, see "Compare Instructions."

Indicators

All indicators are unchanged.

BLGT Example

```
BLGT (R2)
```

In this example, assume that the previous instruction was a compare instruction whose result was *logically greater than*. BLGT causes a branch to the address in register 2.

Branch on Logically Greater Than or Equal (BLGE)

This instruction tests the indicator settings that result from a previous instruction, such as a compare, for a *logically greater than* or *equal* condition. If either condition is met, BLGE causes a branch to *longaddr*. If neither condition is met, the branch is not taken, and the next sequential instruction is executed.

Name	*Operation*	*Operand*
[label]	BLGE	longaddr

This instruction actually tests the carry indicator; if it is off, the branch is taken. For more information about how the indicator is set, see "Compare Instructions."

Indicators

All indicators are unchanged.

BLGE Example

```
BLGE (R2,LOC1)
```

In this example, assume that the previous instruction set a condition of *equal*. BLGE causes a branch to the location whose address is the contents of register 2, increased by the value of LOC1.

Branch on Logically Less Than (BLLT)

This instruction tests the indicator settings that result from a previous instruction, such as a compare, for a *logically less than* condition. If the condition is met, BLLT causes a branch to *longaddr*. If the condition is not met, the branch is not taken, and the next sequential instruction is executed.

Name	*Operation*	*Operand*
[label]	BLLT	longaddr

Note. This instruction actually tests the carry indicator; if it is on, the branch is taken. For more information about how the indicator is set, see "Compare Instructions."

Indicators

All indicators are unchanged.

BLLT Example

```
BLLT LESS
```

In this example, assume that the previous instruction set a condition of *equal*. BLLT does *not* cause a branch to LESS.

Branch on Logically Less Than or Equal (BLLE)

This instruction tests the indicator settings that result from a previous instruction, such as a compare, for a *logically less than* or *equal* condition. If either condition is met, BLLE causes a branch to *longaddr*. If neither condition is met, the branch is not taken, and the next sequential instruction is executed.

Name	*Operation*	*Operand*
[label]	BLLE	longaddr

Note. This instruction actually tests the carry and zero indicators. For more information about how the indicators are set, see "Compare Instructions."

Indicators

All indicators are unchanged.

BLLE Example

```
BLLE (R6)*
```

In this example, assume that the previous instruction set a condition of *logically less than*. BLLE branches to the address defined by the contents of the storage location whose address is in register 6.

Branch on Negative (BN)

This instruction tests the negative result indicator. If it is on, BN causes a branch to *longaddr*. If the indicator is off, the branch is not taken, and the next sequential instruction is executed.

Name	*Operation*	*Operand*
[label]	BN	longaddr

Indicators

All indicators are unchanged.

BN Example

```
BN LOC3+6
```

In this example, assume that the previous instruction turned off the negative result indicator. BN does *not* cause a branch to the location that is 6 bytes past LOC3.

Branch on No Carry (BNCY)

This instruction tests the carry indicator. If it is off, BNCY causes a branch to *longaddr*. If the indicator is on, the branch is not taken, and the next sequential instruction is executed.

Name	*Operation*	*Operand*
[label]	BNCY	longaddr

Indicators

All indicators are unchanged.

BNCY Example

```
BNCY NOCARRY
```

In this example, assume that the previous instruction left the carry indicator off. BNCY causes a branch to NOCARRY.

Branch on Not Condition (BNC)

This instruction tests a condition that you specify. If the condition is met, BNC causes a branch to *longaddr*. If the condition is not met, the branch is not taken, and the next sequential instruction is executed.

Name	*Operation*	*Operand*
[label]	BNC	cond, longaddr

Code BNC to test the indicator settings that result from a previous instruction. For the *cond* operand, code the value of the condition you want to test:

Condition Value	*Condition*
0	Non-zero or non-equal
1	Not positive
2	Not negative
3	Not even
4	Arithmetically greater than or equal
5	Arithmetically greater than
6	Logically greater than
7	Logically greater than or equal (no carry)

Note. BNC causes a branch if the specified condition is *met.*

Indicators

All indicators are unchanged.

BNC Example

```
BNC 6,GREATER
```

In this example, assume that the previous instruction set an *equal* condition. BNC does *not* cause a branch to GREATER.

Branch on Not Condition Code (BNCC)

This instruction tests the even, carry, and overflow indicators. If the tested condition code is not met, BNCC causes a branch to *longaddr.* If the condition code is met, the branch is not taken, and the next sequential instruction is executed.

Name	*Operation*	*Operand*
[label]	BNCC	cond, longaddr

Code BNCC to test the indicator settings that result from a previous instruction. For the *cond* operand, specify the condition code you want to test:

Condition Code	*Indicators*
0	E = 0, C = 0, V = 0
1	E = 0, C = 0, V = 1
2	E = 0, C = 1, V = 0
3	E = 0, C = 1, V = 1
4	E = 1, C = 0, V = 0
5	E = 1, C = 0, V = 1
6	E = 1, C = 1, V = 0
7	E = 1, C = 1, V = 1

The abbreviations used for the indicators are:

- E—even
- C—carry
- V—overflow

Indicators

All indicators are unchanged.

BNCC Example

```
BNCC 3,(R3)
```

In this example, assume that the previous instruction left the even and carry indicators off. BNCC causes a branch to the address in register 3.

Branch on Not Equal (BNE)

This instruction tests the indicator settings that result from a previous instruction, such as a compare, for an *equal* condition. If the condition is not met, BNE causes a branch to *longaddr*. If the condition is met, the branch is not taken, and the next sequential instruction is executed.

Name	*Operation*	*Operand*
[label]	BNE	longaddr

Note. This instruction actually tests the zero indicator.

Indicators

All indicators are unchanged.

BNE Example

```
BNE UNEQUAL
```

In this example, assume that the previous instruction left an *equal* condition. BNE does *not* cause a branch to UNEQUAL.

Branch on Not Error (BNER)

This instruction tests the condition code (after an I/O operation) for an error condition. If there is no error, BNER causes a branch to *longaddr*.

Name	*Operation*	*Operand*
[label]	BNER	longaddr

Note. Coding this instruction does the same thing as coding the BCC instruction to branch on condition code 7.

Indicators

All indicators are unchanged.

Branch on Not Even (BNEV)

This instruction tests the even result indicator. If a previous instruction left it off, BNEV causes a branch to *longaddr*. If the indicator is on, the branch is not taken, and the next sequential instruction is executed.

Name	*Operation*	*Operand*
[label]	BNEV	longaddr

Indicators

All indicators are unchanged.

BNEV Example

```
BNEV (R6)*
```

In this example, assume that BNEV found the even indicator off. BNEV causes a branch to the address defined by the contents of the location whose address is in register 6.

Branch on Not Negative (BNN)

This instruction tests the negative result indicator. If it is off, BNN causes a branch to *longaddr*. If the indicator is on, the branch is not taken, and the next sequential instruction is executed.

Name	*Operation*	*Operand*
[label]	BNN	longaddr

Indicators

All indicators are unchanged.

BNN Example

```
BNN LOC3+6
```

In this example, assume that the previous instruction turned off the negative indicator. BNN causes a branch to the location that is 6 bytes past LOC3.

Branch on Not Overflow (BNOV)

This instruction tests the overflow indicator. If it is off, BNOV causes a branch to *longaddr*. If the indicator is on, the branch is not taken, and the next sequential instruction is executed.

Name	*Operation*	*Operand*
[label]	BNOV	longaddr

Indicators

All indicators are unchanged.

BNOV Example

```
BNOV (R2)
```

In this example, assume that the previous instruction turned on the overflow indicator. BNOV does *not* cause a branch to the address in register 2.

Branch on Not Positive (BNP)

This instruction tests the indicator settings that result from a previous instruction for a *positive* condition. If the condition is not met, BNP causes a branch to *longaddr*. If the condition is met, the branch is not taken, and the next sequential instruction is executed.

Name	*Operation*	*Operand*
[label]	BNP	longaddr

Note. This instruction actually tests the negative and zero result indicators.

Indicators

All indicators are unchanged.

BNP Example

```
BNP LOC5-8
```

In this example, assume that the previous instruction turned on the negative indicator. BNP causes a branch to the location that is 8 bytes before LOC5.

Branch on Not Zero (BNZ)

This instruction tests the zero result indicator. If it is off, BNZ causes a branch to *longaddr*. If the indicator is on, the branch is not taken, and the next sequential instruction is executed.

Name	*Operation*	*Operand*
[label]	BNZ	longaddr

Indicators

All indicators are unchanged.

BNZ Example

```
BNZ (R5)*
```

In this example, assume that the previous instruction turned off the zero indicator. BNZ causes a branch to the address defined by the contents of the location whose address is in register 5.

Branch on Overflow (BOV)

This instruction tests the overflow indicator. If it is on, BOV causes a branch to *longaddr*. If it is off, the branch is not taken, and the next sequential instruction is executed.

Name	*Operation*	*Operand*
[label]	BOV	longaddr

Indicators

All indicators are unchanged.

BOV Example

```
BOV OVER
```

In this example, assume that the previous instruction turned on the overflow indicator. BOV causes a branch to OVER.

Branch on Positive (BP)

This instruction tests the indicator settings that result from a previous instruction for a *positive* condition. If the condition is met, BP causes a branch to *longaddr*. If the condition is not met, the branch is not taken, and the next sequential instruction is executed.

Name	*Operation*	*Operand*
[label]	BP	longaddr

Note. This instruction actually tests the negative and zero result indicators.

Indicators

All indicators are unchanged.

BP Example

```
BP LOC5-4
```

In this example, assume that the previous instruction turned on the negative indicator. BP does *not* cause a branch to the location that is 4 bytes before LOC5.

Branch on Zero (BZ)

This instruction tests the zero result indicator. If it is on, BZ causes a branch to *longaddr*. If the indicator is off, the branch is not taken, and the next sequential instruction is executed.

Name	*Operation*	*Operand*
[label]	BZ	longaddr

Indicators

All indicators are unchanged.

BZ Example

```
BZ ZERO
```

In this example, assume that the previous instruction turned off the zero indicator. BZ does *not* cause a branch to ZERO.

No Operation (NOP)

This instruction causes an unconditional branch to the next sequential instruction.

Name	*Operation*	*Operand*
[label]	NOP	

Indicators

All indicators are unchanged.

Coding Jump Instructions

You can jump to locations that are within the same CSECT as the jump instruction. For all jump instructions, code either the *jdisp* or *jaddr* operand to specify the address to be jumped to. This address must be within −256 to 254 bytes of the byte following the jump instruction.

If you use the *jdisp* form, code—as an even absolute value or expression—a displacement from the byte following the jump instruction.

If you use the *jaddr* form, code—as a relocatable expression—the even byte address you want to jump to.

Note. The IAR points to the byte following the jump instruction; therefore, *jdisp* is actually a displacement from the IAR. Jump instructions are the only IAR-relative instructions.

Jump (J)

This instruction causes an unconditional jump to the specified address.

Name	*Operation*	*Operand*
[label]	J	jdisp jaddr

Indicators

All indicators are unchanged.

J Example

```
J THERE
```

This instruction causes a jump to the location THERE.

Jump and Link (JAL)

This instruction saves—in a register—the address of the next sequential instruction, then causes a jump to the specified location.

Name	*Operation*	*Operand*
[label]	JAL	jdisp jaddr

For *reg* specify the register in which you want to save the address of the next sequential instruction.

Indicators

All indicators are unchanged.

JAL Example

```
JAL 8,R7
```

In this example, JAL saves—in register 7—the address of the instruction that follows this JAL. This instruction then causes a jump to the location that is 8 bytes past the byte that follows this JAL instruction.

Jump if Mixed (JMIX)

After a Test Word Immediate (TWI) instruction, JMIX causes a jump if the bits tested by TWI are a combination of zeros and ones.

Name	*Operation*	*Operand*
[label]	JMIX	jdisp jaddr

Note. JMIX actually tests the zero and negative indicators.

Indicators

All indicators are unchanged.

JMIX Example

Assume that the word at location TEST contains X'0369'.

```
TWI   X'Ø369',TEST
JMIX  MIXED
```

Because the bits tested by TWI are all ones, JMIX does *not* cause a jump to location MIXED.

Jump if Not Mixed (JNMIX)

After a Test Word Immediate (TWI) instruction, JNMIX causes a jump if the bits tested by TWI are either all zeros or all ones.

Note. JNMIX actually tests the zero and negative indicators.

Name	*Operation*	*Operand*
[label]	JNMIX	jdisp jaddr

Note. If the first operand (the 1-word mask) of the TWI instruction is all zeros, the resulting condition is *not mixed.* In this case, JNMIX causes a jump.

Indicators

All indicators are unchanged.

JNMIX Example

Assume that R4 contains X'0000'.

```
TWI   X'ØØFF',R4
JNMIX 14
```

Because the bits tested by TWI are all zeros, JNMIX causes a jump to the location that is 14 bytes past the byte following this JNMIX.

Jump if Not Off (JNOFF)

After a Test Bit (TBT) or Test Word Immediate (TWI) instruction, JNOFF causes a jump if:

- The bit tested by TBT is on, or

- The bits tested by TWI are either mixed or all on.

Name	*Operation*	*Operand*
[label]	JNOFF	jdisp jaddr

Note. JNOFF actually tests the zero indicator.

Indicators

All indicators are unchanged.

JNOFF Example

Assume that the word whose address is in R3 contains X'03AC'.

```
TWI   X'ØC53',(R3)
JNOFF 36
```

Because the tested bits are all off, JNOFF does *not* cause a jump to the location that is 36 bytes past the byte following this JNOFF.

Jump if Not On (JNON)

After a Test Bit (TBT) or Test Word Immediate (TWI) instruction, JNON causes a jump if:

- The bit tested by TBT is off, or
- The bits tested by TWI are either mixed or all off.

Note. JNON actually tests the negative indicator.

Name	*Operation*	*Operand*
[label]	JNON	jdisp jaddr

Note. If the first operand (the 1-word mask) of a TWI instruction is all zeros, the resulting condition is *not on*. In this case, JNON causes a jump.

Indicators

All indicators are unchanged.

JNON Example

Assume that the word in R7 contains X'0123'.

```
TWI   X'Ø321',R7
JNON  -6
```

Because the tested bits are mixed, JNON causes a jump to the location that is 6 bytes before the byte following this JNON.

Jump if Off (JOFF)

After a Test Bit (TBT) or Test Word Immediate (TWI) instruction, JOFF causes a jump if:

- The bit tested by TBT is off, or
- The bits tested by TWI are all off.

Note. JOFF actually tests the zero indicator.

Name	*Operation*	*Operand*
[label]	JOFF	jdisp jaddr

Note. If the first operand (the 1-word mask) of a TWI instruction is all zeros, the resulting condition is *off*. In this case, JOFF causes a jump.

Indicators

All indicators are unchanged.

JOFF Example

```
TBT   (R6,5)
JOFF  OFF
```

Assume that the byte whose address is in R6 contains:
1111 0000
Because the sixth bit is off, JOFF causes a jump to location OFF.

Jump if On (JON)

After a Test Bit (TBT) or Test Word Immediate (TWI) instruction, JON causes a jump if:

- The bit tested by TBT is on, or
- The bits tested by TWI are all on.

Name	*Operation*	*Operand*
[label]	JON	jdisp jaddr

Note. JON actually tests the negative indicator.

Indicators

All indicators are unchanged.

JON Example

```
TBT (R1,Ø)
JON ON+4
```

Assume that the byte whose address is in R1 contains:
0000 1111
Because the first bit is off, JON does *not* cause a jump to the address that is 4 bytes past location ON.

Jump on Carry (JCY)

This instruction tests the carry indicator. If the indicator is on, JCY jumps to the specified location. If the indicator is off, the jump is not taken, and the next sequential instruction is executed.

Name	*Operation*	*Operand*
[label]	JCY	jdisp jaddr

Indicators

All indicators are unchanged.

JCY Example

```
JCY CARRY-4
```

In this example, assume that JCY found the carry indicator on. JCY jumps to the location that is 4 bytes before the address CARRY.

Jump on Condition (JC)

This instruction tests a condition that you specify. If the tested condition is met, JC causes a jump to the specified location. If the condition is not met, the jump is not taken, and the next sequential instruction is executed.

Name	*Operation*	*Operand*
[label]	JC	jdisp jaddr

Code the JC instruction to test the indicator settings that result from a previous instruction. For the *cond* operand, code the value of the condition you want to test:

Condition Value	*Condition*
0	Zero or equal
1	Positive and non-zero
2	Negative
3	Even
4	Arithmetically less than
5	Arithmetically less than or equal
6	Logically less than or equal
7	Logically less than (carry)

Indicators

All indicators are unchanged.

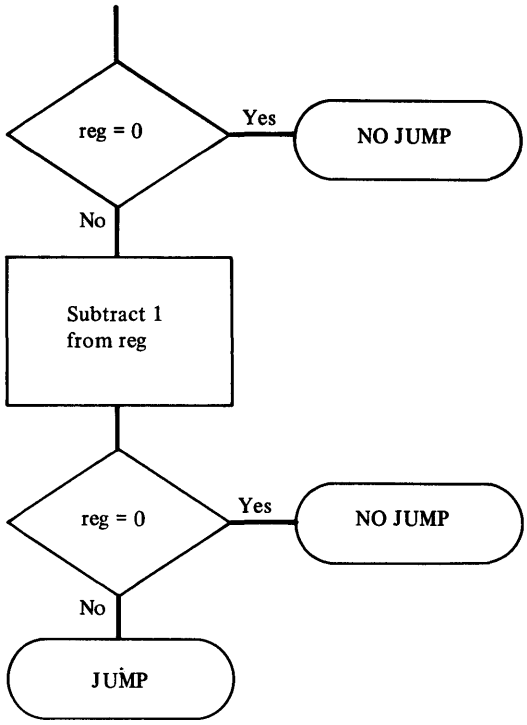
JC Example

```
JC 2,1Ø
```

In this example assume that a previous instruction set the negative result indicator on. JC causes a jump to the location that is 10 bytes past the byte following this JC instruction.

Jump on Count (JCT)

This instruction tests the contents of the specified register. If the contents of the register are not zero, JCT decreases the register by 1. If the contents are still not zero, JCT causes a jump to the specified location.



Name	*Operation*	*Operand*
[label]	JCT	jdisp, reg jaddr, reg

Indicators

All indicators are unchanged.

JCT Example

```
JCT ZERO,R3
```

Assume that R3 (before the decrement) contains X'0025'. JCT decreases R3 by 1, leaving X'0024', and causes a jump to ZERO.

Jump on Equal (JE)

This instruction tests the result of a previous instruction, such as a compare, for an *equal* condition. If the condition is met, JE causes a jump to the specified location. If the condition is not met, no jump is taken, and the next sequential instruction is executed.

Name	*Operation*	*Operand*
[label]	JE	jdisp jaddr

Note. This instruction actually tests the zero result indicator.

Indicators

All indicators are unchanged.

JE Example

```
JE EQUAL
```

Assume that this JE was preceded by a compare instruction whose result was *equal.* JE causes a jump to EQUAL.

Jump on Even (JEV)

This instruction tests the even indicator. If a previous instruction left it on, JEV causes a jump to the specified location. If the indicator is off, the jump is not taken, and the next sequential instruction is executed.

Name	*Operation*	*Operand*
[label]	JEV	jdisp jaddr

Indicators

All indicators are unchanged.

JEV Example

```
JEV 26
```

In this example assume that the previous instruction left the even indicator on. JEV causes a jump to the location that is 26 bytes past the byte following this JEV.

Jump on Greater Than (JGT)

This instruction tests the result of a logical instruction for an arithmetically *greater than* condition. If the condition is met, JGT causes a jump to the specified location. If the condition is not met, the jump is not taken, and the next sequential instruction is executed.

Name	*Operation*	*Operand*
[label]	JGT	jdisp jaddr

Note. This instruction actually tests the negative, overflow, and zero result indicators.

Indicators

All indicators are unchanged.

JGT Example

```
JGT GREATER+6
```

In this example, assume that the previous instruction left a *greater than* condition. JGT causes a jump to the location that is 6 bytes past GREATER.

Jump on Greater Than or Equal (JGE)

This instruction tests the result of a logical instruction for an arithmetically *greater than* or an *equal* condition. If either condition is met, JGE causes a jump to the specified location. If neither condition is met, the jump is not taken, and the next sequential instruction is executed.

Name	*Operation*	*Operand*
[label]	JGE	jdisp jaddr

Note. This instruction actually tests the negative and overflow indicators.

Indicators

All indicators are unchanged.

JGE Example

```
JGE 84
```

In this example, assume that the previous instruction left a *less than* condition. JGE does *not* cause a jump to the location that is 84 bytes past the byte that follows this JGE instruction.

Jump on Less Than (JLT)

This instruction tests the result of a logical instruction for an arithmetically *less than* condition. If the condition is met, JLT causes a jump to the specified location. If the condition is not met, the jump is not taken, and the next sequential instruction is executed.

Name	*Operation*	*Operand*
[label]	JLT	jdisp jaddr

Note. This instruction actually tests the negative and overflow indicators.

Indicators

All indicators are unchanged.

JLT Example

```
JLT LESS+2
```

In this example, assume that the previous instruction left a *less than* condition. JLT causes a jump to the location that is 2 bytes past LESS.

Jump on Less Than or Equal (JLE)

This instruction tests the result of a logical instruction for an arithmetically *less than* or an *equal* condition. If either condition is met, JLE causes a jump to the specified location. If neither condition is met, the jump is not taken, and the next sequential instruction is executed.

Name	*Operation*	*Operand*
[label]	JLE	jdisp jaddr

Note. This instruction actually tests the overflow, negative, and zero result indicators.

Indicators

All indicators are unchanged.

JLE Example

```
JLE THERE-4
```

In this example, assume that the previous instruction left an *equal* condition. JLE causes a jump to the location that is 4 bytes before address THERE.

Jump on Logically Greater Than (JLGT)

This instruction tests the result of a logical instruction, such as a compare, for a *logically greater than* condition. If the condition is met, JLGT causes a jump to the specified location. If the condition is not met, the jump is not taken, and the next sequential instruction is executed.

Name	*Operation*	*Operand*
[label]	JLGT	jdisp jaddr

Note. This instruction actually tests the carry and zero indicators. If both indicators are off, the jump is taken. For more information about how the indicators are set, see "Logical Instructions."

Indicators

All indicators are unchanged.

JLGT Example

```
JLGT THERE
```

In this example, assume that the previous instruction was a compare instruction whose result was *logically greater than.* JLGT causes a jump to location THERE.

Jump on Logically Greater Than or Equal (JLGE)

This instruction tests the result of a logical or arithmetic instruction, such as a compare or subtract, for a *logically greater than* or *logically equal* condition. If either condition is met, JLGE causes a jump to the specified location. If neither condition is met, the jump is not taken, and the next sequential instruction is executed.

Name	*Operation*	*Operand*
[label]	JLGE	jdisp jaddr

Note. This instruction actually tests the carry indicator; if it is off, the jump is taken. For more information about how the indicator is set, see "Logical Instructions" and "Arithmetic Instructions."

Indicators

All indicators are unchanged.

JLGE Example

```
JLGE -12
```

Assume that this JLGE appears in this piece of code:

```
X MVWI 14,R2
  MVA  LOC,R1
  CW   R3,R4
  JLGE -12
  CMR  R1,R2
```

If CW R3,R4 set a condition of *logically greater than or equal,* JLGE −12 causes a jump to MVWI 14,R2. (JLGE X would do the same thing.)

Jump on Logically Less Than (JLLT)

This instruction tests the result of a logical instruction, such as a compare, for a *logically less than* condition. If the condition is met, JLLT causes a jump to the specified location. If the condition is not met, the jump is not taken, and the next sequential instruction is executed.

Name	*Operation*	*Operand*
[label]	JLLT	jdisp jaddr

Note. This instruction actually tests the carry indicator—if it is on, the jump is taken. For more information about how the indicator is set, see "Logical Instructions."

Indicators

All indicators are unchanged.

JLLT Example

```
JLLT LESS
```

In this example assume that the previous instruction set a condition of *equal.* JLLT does not cause a jump to LESS.

Jump on Logically Less Than or Equal (JLLE)

This instruction tests the result of a logical instruction, such as a compare, for a *logically less than* or *equal* condition. If either condition is met, JLLE causes a jump to the specified location. If neither condition is met, the jump is not taken, and the next sequential instruction is executed.

Name	*Operation*	*Operand*
[label]	JLLE	jdisp jaddr

Note. This instruction actually tests the carry and zero indicators. Both indicators must be on for the jump to be taken. For more information about how the indicators are set, see "Logical Instructions."

Indicators

All indicators are unchanged.

JLLE Example

```
JLLE LESS
```

In this example, assume that the previous instruction set a condition of *logically less than.* JLLE causes a jump to LESS.

Jump on Negative (JN)

This instruction tests the negative result indicator. If it is on, JN causes a jump to the specified location. If the indicator is off, the jump is not taken, and the next sequential instruction is executed.

Name	*Operation*	*Operand*
[label]	JN	jdisp jaddr

Indicators

All indicators are unchanged.

JN Example

```
JN LOC3+6
```

In this example, assume that the previous instruction turned off the negative result indicator. JN does *not* cause a jump to the location that is 6 bytes past LOC3.

Jump on No Carry (JNCY)

This instruction tests the carry indicator. If it is off, JNCY causes a jump to the specified location. If the indicator is on, the jump is not taken, and the next sequential instruction is executed.

Name	*Operation*	*Operand*
[label]	JNCY	jdisp jaddr

Indicators

All indicators are unchanged.

JNCY Example

```
JNCY NOCARRY
```

In this example assume that the previous instruction left the carry indicator off. JNCY causes a jump to NOCARRY.

Jump on Not Condition (JNC)

This instruction tests a condition that you specify. If the condition is met, JNC causes a jump to the specified location. If the condition is not met, the jump is not taken, and the next sequential instruction is executed.

Name	*Operation*	*Operand*
[label]	JNC	cond, jdisp cond, jaddr

Code JNC to test the result of a previous instruction. For the *cond* operand, code the value of the condition you want to test:

Condition Value	*Condition*
0	Non-zero or non-equal
1	Not positive
2	Not negative
3	Not even
4	Arithmetically greater than or equal
5	Arithmetically greater than
6	Logically greater than
7	Logically greater than or equal (no carry)

Note. JNC causes a branch if the specified condition is *met.*

Indicators

All indicators are unchanged.

JNC Example

```
JNC 6,-12
```

In this example, assume that the previous instruction set a *logically equal* condition. JNC does *not* cause a jump to the location that is 12 bytes before the byte that follows the JNC instruction.

Jump on Not Equal (JNE)

This instruction tests the result of a previous instruction, such as a compare, for an *equal* condition. If the condition is not met, JNE causes a jump to the specified location. If the condition is met, the jump is not taken, and the next

sequential instruction is executed.

Name	*Operation*	*Operand*
[label]	JNE	jdisp jaddr

Note. This instruction actually tests the zero result indicator.

Indicators

All indicators are unchanged.

JNE Example

```
JNE UNEQUAL
```

In this example, assume that the previous instruction left an *equal* condition. JNE does *not* cause a jump to UNEQUAL.

Jump on Not Even (JNEV)

This instruction tests the even result indicator. If a previous instruction left it off, JNEV causes a jump to the specified location. If the indicator is on, the jump is not taken, and the next sequential instruction is executed.

Name	*Operation*	*Operand*
[label]	JNEV	jdisp jaddr

Indicators

All indicators are unchanged.

JNEV Example

```
JNEV 6
```

In this example, assume that JNEV found the even indicator off. JNEV causes a jump to the location that is 6 bytes past the byte following this JNEV instruction.

Jump on Not Negative (JNN)

This instruction tests the negative result indicator. If it is off, JNN causes a jump to the specified location. If the indicator is on, the jump is not taken, and the next sequential instruction is executed.

Name	*Operation*	*Operand*
[label]	JNN	jdisp jaddr

Indicators

All indicators are unchanged.

JNN Example

```
JNN LOC3-6
```

In this example assume that the previous instruction turned off the negative indicator. JNN causes a jump to the location that is 6 bytes before LOC3.

Jump on Not Positive (JNP)

This instruction tests the result of a previous instruction for a *positive* condition. If the condition is not met, JNP causes a jump to the specified location. If the condition is met, the jump is not taken, and the next sequential instruction is executed.

Name	*Operation*	*Operand*
[label]	JNP	jdisp jaddr

Note. This instruction actually tests the negative and zero result indicators.

Indicators

All indicators are unchanged.

JNP Example

```
JNP THERE
```

In this example assume that the previous instruction turned off the negative and zero indicators (leaving a *positive* condition). JNP does *not* cause a jump to THERE.

Jump on Not Zero (JNZ)

This instruction tests the zero result indicator. If it is off, JNZ causes a jump to the specified location. If the indicator is on, the jump is not taken, and the next sequential instruction is executed.

Name	*Operation*	*Operand*
[label]	JNZ	jdisp jaddr

Indicators

All indicators are unchanged.

JNZ Example

```
JNZ -4
```

In this example assume that the previous instruction turned off the zero indicator. JNZ causes a jump to the location that is 4 bytes before the byte following this JNZ instruction.

Jump on Positive (JP)

This instruction tests the result of a previous instruction for a *positive* condition. If the condition is met, JP causes a jump to the specified location. If the condition is not met, the jump is not taken, and the next sequential instruction is executed.

Name	*Operation*	*Operand*
[label]	JP	jdisp jaddr

Note. This instruction actually tests the negative and zero result indicators.

Indicators

All indicators are unchanged.

JP Example

```
JP LOC5-12
```

In this example assume that the previous instruction turned on the negative indicator. JP does *not* cause a jump to the location that is 12 bytes before LOC5.

Jump on Zero (JZ)

This instruction tests the zero result indicator. If it is on, JZ causes a jump to the specified location. If the indicator is off, the jump is not taken, and the next sequential instruction is executed.

Name	*Operation*	*Operand*
[label]	JZ	jdisp jaddr

Indicators

All indicators are unchanged.

JZ Example

```
JZ ZERO
```

In this example, assume that the previous instruction turned off the zero indicator. JZ does *not* cause a jump to ZERO.

Shift Instructions

Coding Shift Instructions

The shift instructions all have the same basic syntax. The first operand is always the shift count, and the second is always the register to be shifted.

You can code shift count as an absolute value or expression, or you can code it in register form, where bits 8–15 of the register contain the count. If the shift count is in a register, that register is not altered by the shift instruction, unless the instruction is SLT or SLTD, or the same register is also specified as the register to be shifted.

Note. For SLT and SLTD, code the shift count in register form *only.*

If you code a shift count value, it can be in the range 0–16 (for *cnt16*) or 0–31 (for *cnt31*). You may code a value greater than 16 for *cnt16*, however, it will be flagged with a warning message. When the instruction is executed, a shift count greater than 16 will lengthen the execution time. If you code a register that contains the shift count, the count can be in the range 0–255. Note that if you code a shift count of 0, the register is not shifted.

The second operand is the register to be shifted. In the case of double shift instructions, the register you code here is the first register of a register pair. For example, if you code R5, the instruction shifts the pair R5,R6. If you code R7, the instruction shifts the pair R7,R0.

Shift Left Circular (SLC)

This instruction shifts the contents of a register to the left by a specified number of bits. The bits shifted out of the high-order bit (bit 0) reenter in the low-order bit (bit 15).

Name	*Operation*	*Operand*
[label]	SLC	cnt16, reg reg, reg

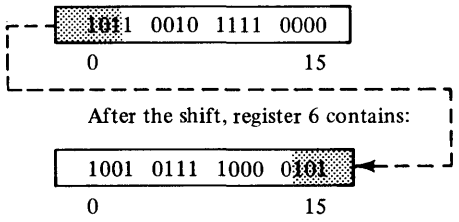
Indicators

The carry and overflow indicators are unchanged. The remaining indicators reflect the final contents of the register.

SLC Example

```
SLC 3,R6
```

Before the shift, register 6 contains:



Shift Left Circular Double (SLCD)

This instruction shifts the contents of a register pair to the left by a specified number of bits. The bits shifted out of the high-order bit (bit 0) reenter in the low-order bit (bit 31).

Name	*Operation*	*Operand*
[label]	SLCD	cnt31, reg reg, reg

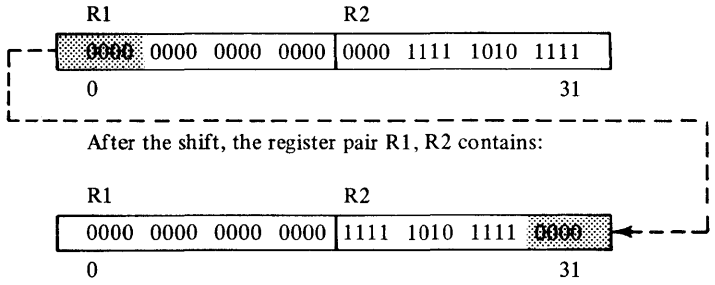
Indicators

The carry and overflow indicators are unchanged. The remaining indicators reflect the final contents of the register pair.

SLCD Example

```
SLCD 4,R1
```

Before the shift, the register pair R1, R2 contains:



Shift Left Logical (SLL)

This instruction shifts the contents of a register to the left by a specified number of bits. The vacated low-order bits are filled with zeros.

Name	*Operation*	*Operand*
[label]	SLL	cnt16, reg reg, reg

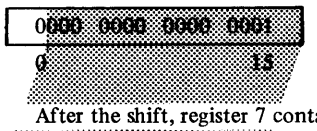
Indicators

The overflow indicator is first reset, then set to 1 if the most significant bit in the register changed during the shift. The carry indicator reflects the last bit shifted out of bit 0. The remaining indicators reflect the final contents of the register.

SLL Example

```
SLL 1,R7
```

Before the shift, register 7 contains:



After the shift, register 7 contains:

Shift Left Logical Double (SLLD)

This instruction shifts the contents of a register pair to the left by a specified number of bits. The vacated low-order bits are filled with zeros.

Name	*Operation*	*Operand*
[label]	SLLD	cnt31, reg reg, reg

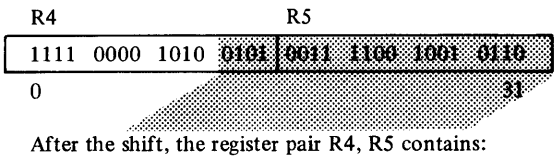
Indicators

The overflow indicator is first reset, then set to 1 if the most significant bit in the register pair changed during the shift. The carry indicator reflects the last bit shifted out of bit 0. The remaining indicators reflect the final contents of the register pair.

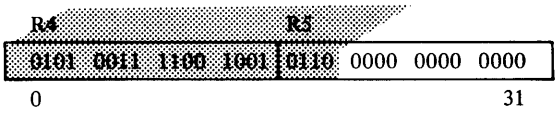
SLLD Example

```
SLLD R7,R4
```

Assume that register 7 contains X'000C'.
Before the shift, the register pair R4, R5 contains:



After the shift, the register pair R4, R5 contains:



Shift Left and Test (SLT)

This instruction shifts the contents of a register to the left. It continues shifting until it has:

- Shifted the number of bits specified as a shift count, or
- Shifted a 1-bit out of bit zero.

The vacated low-order bits are filled with zeros.

Name	*Operation*	*Operand*
[label]	SLT	reg, reg

If SLT shifts a 1-bit out of bit zero *before* it has shifted the number of bits you specified, the remaining shift count is loaded into bits 8–15 of the register you coded for the first operand.

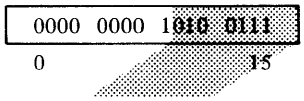
Indicators

The overflow and carry indicators are first reset. Then the overflow indicator is set to 1 if the most significant bit in the register changed during the shift. The carry indicator reflects the last bit shifted out of bit 0. The remaining indicators reflect the contents of the register you coded for the first operand.

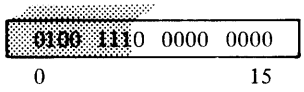
SLT Example

```
SLT R5,R2
```

In this example, assume that register 5 contains X'000C'.
Before the shift, register 2 contains:



After the shift, register 2 contains:



and register 5 contains X'0003', the shift count that remained after SLT shifted a 1-bit out of the high-order bit of the register.

Shift Left and Test Double (SLTD)

This instruction shifts the contents of a register pair to the left. It continues shifting until it has:

- Shifted the number of bits specified as a shift count, OR
- Shifted a 1-bit out of bit zero of the first register in the pair.

The vacated low-order bits are filled with zeros.

Name	*Operation*	*Operand*
[label]	SLTD	reg, reg

If SLTD shifts a 1-bit out of bit zero *before* it has shifted the number of bits you specified, the remaining shift count is loaded into bits 8–15 of the register you coded for the first operand.

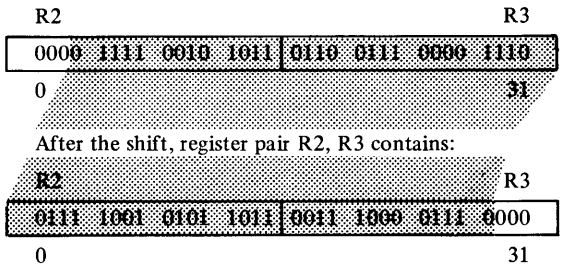
Indicators

The carry and overflow indicators are first reset. Then the overflow indicator is set to 1 if the most significant bit in the register pair changed during the shift. The carry indicator reflects the last bit shifted out of bit 0. The remaining indicators reflect the contents of the register you coded for the first operand.

SLTD Example

```
SLTD R1,R2
```

In this example, assume that register 1 contains X'0003'.
Before the shift, register pair R2, R3 contains:



After the shift, register pair R2, R3 contains:

and R1 contains 0.

Shift Right Arithmetic (SRA)

This instruction shifts the contents of a register to the right by a specified number of bits. The original high-order bit of the register is propagated through the vacated high-order bits.

Name	*Operation*	*Operand*
[label]	SRA	cnt16, reg reg, reg

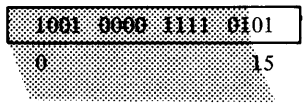
Indicators

The carry and overflow indicators are unchanged. The remaining indicators reflect the final contents of the register.

SRA Example

```
SRA 2,R5
```

Before the shift, register 5 contains:



After the shift, register 5 contains:

Shift Right Arithmetic Double (SRAD)

This instruction shifts the contents of a register pair to the right by a specified number of bits. The original high-order bit of the register pair is propagated through the vacated high-order bits.

Name	*Operation*	*Operand*
[label]	SRAD	cnt31, reg reg, reg

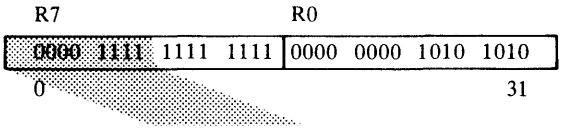
Indicators

The carry and overflow indicators are unchanged. The remaining indicators reflect the final contents of the register pair.

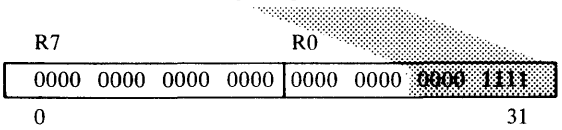
SRAD Example

```
SRAD R1,R7
```

Assume that register 1 contains X'0018'.
Before the shift, the register pair R7, R0 contains:



After the shift, the register pair R7, R0 contains:



Shift Right Logical (SRL)

This instruction shifts the contents of a register to the right by a specified number of bits. The vacated high-order bits of the register are filled with zeros.

Name	*Operation*	*Operand*
[label]	SRL	cnt16, reg reg, reg

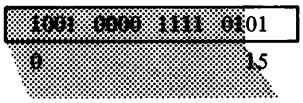
Indicators

The carry and overflow indicators are unchanged. The remaining indicators reflect the final contents of the register.

SRL Example

```
SRL 2,R5
```

Before the shift, register 5 contains:



After the shift, register 5 contains:

Shift Right Logical Double (SRLD)

This instruction shifts the contents of a register pair to the right by a specified number of bits. The vacated high-order bits of the register pair are filled with zeros.

Name	*Operation*	*Operand*
[label]	SRLD	cnt31, reg reg, reg

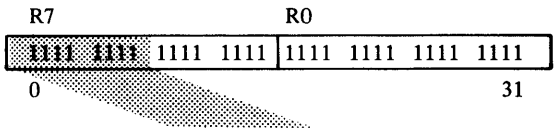
Indicators

The carry and overflow indicators are unchanged. The remaining indicators reflect the final contents of the register pair.

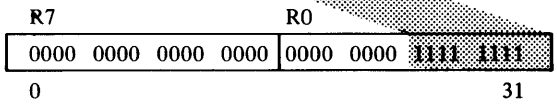
SRLD Example

```
SRLD R1,R7
```

In this example, assume that register 1 contains X'0018'.
Before the shift, the register pair R7, R0 contains:



After the shift, the register pair contains:



Stack Instructions

Store Multiple (STM)

STM saves the contents of one or more general-purpose registers from your main routine. Code it at the beginning of a subroutine.

Name	*Operation*	*Operand*
[label]	STM	reg, addr4[,abcnt]

In the first operand, code a register n. STM stores register 7, then registers 0 through n. For example, if you code register 2, STM stores registers 7, 0, 1, and 2. If you code register 7, STM stores register 7 only.

In the second operand, specify the address of the stack control block that points to the stack where you want the registers stored.

Use the optional third operand to define the size, in bytes, of a work storage area within the stack. For *abcnt*, code the size of the work area. That amount must be an even number in the range 0–16382.

After it stores the registers and reserves the work area, STM loads *reg* (the first operand) with the address of the low-storage end of the work storage area or the address of the last register stored if *abent* was not specified or specified as 0.

Indicators

All indicators are unchanged.

STM Example

```
STM R4,(R1),32
```

- R1 contains the address of a stack control block.
- The stack now contains 32 bytes of work storage and a 2-byte control word, in addition to the 12 bytes required for the registers.

STM stores registers 7, 0, 1, 2, 3, and 4 in the specified stack, then loads register 4 with the address of the work area.

Load Multiple and Branch (LMB)

This instruction is useful when a subroutine passes control back to your main program. At the end of the subroutine, LMB reloads the registers from a stack, then branches to the address in register 7.

Name	*Operation*	*Operand*
[label]	LMB	addr4

Before it gave control to the subroutine, the main program loaded register 7 with the address that will gain control when the subroutine finishes. This address is usually the address of the next sequential instruction following the branch to the subroutine.

After the main program passed control to the subroutine, the subroutine saved the contents of the index registers in a stack (with a STM instruction).

Indicators

All indicators are unchanged.

LMB Example

```
LMB (R1)
```

Assume that register 1 contains the address of the stack control block that points to the stack where the registers are stored. LMB reloads the registers from the stack, and passes control to the address in register 7.

Coding Pop/Push Instructions

In general, a push instruction moves an element from a register to a stack, and a pop instruction moves an element from a stack to a register.

For the stack operand (*addr4*) in a push or pop instruction, code the address of the stack control block that points to the stack you want to use.

For the register operand (*reg*) in a push or pop instruction, code the register you want to use. If you code a doubleword instruction (PD or PSD), the register you code is the first register of a pair. Note that if you code register 7 as the first register of a pair, the instruction uses registers 7 and 0.

Pop Byte (PB)

This instruction moves a byte from a stack and places it into a register.

Name	*Operation*	*Operand*
[label]	PB	addr4, reg

PB moves the top byte in the stack into bits 8–15 of *reg*. Bits 0–7 of *reg* are unchanged.

After PB executes, the Top Element Address pointer in the stack control block points to the next byte to be popped from the stack.

Indicators

All indicators are unchanged.

PB Example

```
PB (R1),R3
```

In this example, register 1 contains the address of a stack control block. PB moves the top byte from the stack into bits 8–15 of register 3. Bits 0–7 of register 3 are unchanged. After this PB executes, the Top Element Address pointer is updated and points to the next byte to be popped.

Pop Doubleword (PD)

This instruction moves a doubleword from a stack and places it into a pair of registers.

Name	*Operation*	*Operand*
[label]	PD	addr4, reg

PD moves the top doubleword in the stack into the register pair specified by *reg*.

After PD executes, the Top Element Address pointer in the stack control block points to the next doubleword to be popped from the stack.

Indicators

All indicators are unchanged.

PD Example

```
PD (R1),R3
```

In this example, register 1 contains the address of a stack control block. PD moves the top doubleword from the stack into registers 3 and 4. After PD executes, the Top Element Address pointer is updated and points to the next doubleword to be popped.

```
PD STACKØ1,R7
```

In this example, STACK01 is the address of a stack control block. PD moves the top doubleword from the stack into registers 7 and 0. After PD executes, the Top Element Address pointer is updated and points to the next doubleword to be popped.

Pop Word (PW)

This instruction moves a word from a stack and places it into a register.

Name	*Operation*	*Operand*
[label]	PW	addr4, reg

PW moves the top word in the stack into *reg*.

After PW executes, the Top Element Address pointer in the stack control block points to the next word to be popped from the stack.

Indicators

All indicators are unchanged.

PW Example

```
PW (R1),R5
```

In this example, register 1 contains the address of a stack control block. PW moves the top word from the stack into register 5. After this PW executes, the Top Element Address pointer is updated and points to the next word to be popped.

Push Byte (PSB)

This instruction moves a byte from a register and places it into a stack.

Name	*Operation*	*Operand*
[label]	PSB	reg, addr4

PSB moves bits 8–15 from the *reg* into the stack; *reg* is unchanged.

After PSB executes, the Top Element Address pointer in the stack control block points to the byte just pushed into the stack.

Indicators

All indicators are unchanged.

PSB Example

```
PSB R2,(R1)
```

In this example, R1 contains the address of a stack control block. PSB pushes bits 8–15 from R2 into the stack. After this PSB executes, the Top Element Address pointer is updated, and points to the byte just pushed into the stack. R2 is unchanged.

Push Doubleword (PSD)

This instruction moves a doubleword from a register pair and places it into a stack.

Name	*Operation*	*Operand*
[label]	PSD	reg, addr4

After PSD executes, the Top Element Address pointer in the stack control block points to the doubleword just pushed into the stack. The register pair is unchanged.

Indicators

All indicators are unchanged.

PSD Example

```
PSD R5,(R1)
```

In this example, R1 contains the address of a stack control block. PSD pushes the contents of registers 5 and 6 into the stack. After this PSD executes, the Top Element Address pointer is updated, and points to the doubleword just pushed into the stack. R5 and R6 are unchanged.

Push Word (PSW)

This instruction moves a word from a register and places it into a stack.

Name	*Operation*	*Operand*
[label]	PSW	reg, addr4

After PSW executes, the Top Element Address pointer in the stack control block points to the word just pushed into the stack. The register is unchanged.

Indicators

All indicators are unchanged.

PSW Example

```
PSW R4,(R1)
```

In this example, R1 contains the address of a stack control block. PSW pushes the contents of R4 into the stack. After this PSW executes, the Top Element Address pointer is updated, and points to the word just pushed into the stack. R4 is unchanged.

Compare Instructions

Using Compare Instructions

For a compare instruction, you code two operands. The operands are compared, and indicators are set to reflect the result. Results of the compare can be tested arithmetically or logically. Both operands remain unchanged.

Note. A compare operation actually subtracts the first operand from the second, then sets indicators to reflect the result. Both operands are unchanged. The result is expressed in terms of the second operand relative to the first; for example, *arithmetically greater than* means that the second operand is greater than the first.

An arithmetic test looks at the *value* of each operand, while a logical test looks for the operand with the most significant bit ON. For example, code a compare with the operands A,B. Assume that the value of A is −31, and the value of B is +57. An *arithmetic* test would look at the values and determine that B>A. A *logical* test would look at the individual bits:

−31 = 1110 0001
+57 = 0011 1001

Because −31 (the A value) has its most significant bits ON, the logical test determines that B<A.

You can interpret a compare instruction as either arithmetic or logical, depending on the indicators you test after it executes. For example, if the operands on a compare instruction are A,B, here is how the indicators are set:

If the result of compare A, B is	*These indicators are set*
(Arithmetic)	
B = A	Z = 1
B ≠ A	Z = 0
B < A	(N = 1, V = 0) or (N = 0, V = 1)
B ≤ A	(N = 1, V = 0) or (N = 0, V = 1) or Z = 1
B > A	[(N = 1, V = 1) or (N = 0, V = 0)] and Z = 0
B ≥ A	(N = 1, V = 1) or (N = 0, V = 0)
(Logical)	
B > A	(C = 0, Z = 0)
B ≥ A	C = 0
B < A	C = 1
B ≤ A	C = 1 or Z = 1

The abbreviations used for the indicators are:

- Z—zero
- N—negative
- V—overflow
- C—carry.

Compare Address (CA)

This instruction compares either a word in a register or a word in storage to an address value, then sets indicators to reflect the result.

Name	*Operation*	*Operand*
[label]	CA	raddr, reg raddr, addr4

Indicators

See "Using Compare Instructions" for a description of the indicator settings.

CA Example

```
CA        FIELD,DATA
```

In this example, assume that the address of FIELD is X'5024', and the word at the storage address defined by DATA contains X'6000'. CA compares these values, and sets *arithmetically greater than* and *logically greater than* conditions.

Compare Byte (CB)

This instructions compares:

- A byte in a register (bits 8–15) to a byte in storage, or
- A byte in storage to a byte in storage,

and sets indicators to reflect the result.

Name	*Operation*	*Operand*
[label]	CB	addr4, reg addr5, addr4

Indicators

See "Using Compare Instructions" for a description of the indicator settings.

CB Example

```
CB BYTE+3,R4
```

Assume that the byte at location BYTE+3 contain X'02' and bits 8–15 of register 4 contain X'FD'. CB compares the two values, and sets *arithmetically less than* and *logically greater than* conditions.

Compare Byte Field Equal and Decrement (CFED)

This instruction compares successive (right-to-left) bytes in one field to corresponding bytes in another field. It compares two bytes at a time until it finds an *equal* condition OR until it has exhausted the length count.

Name	*Operation*	*Operand*
[label]	CFED	(reg), (reg)

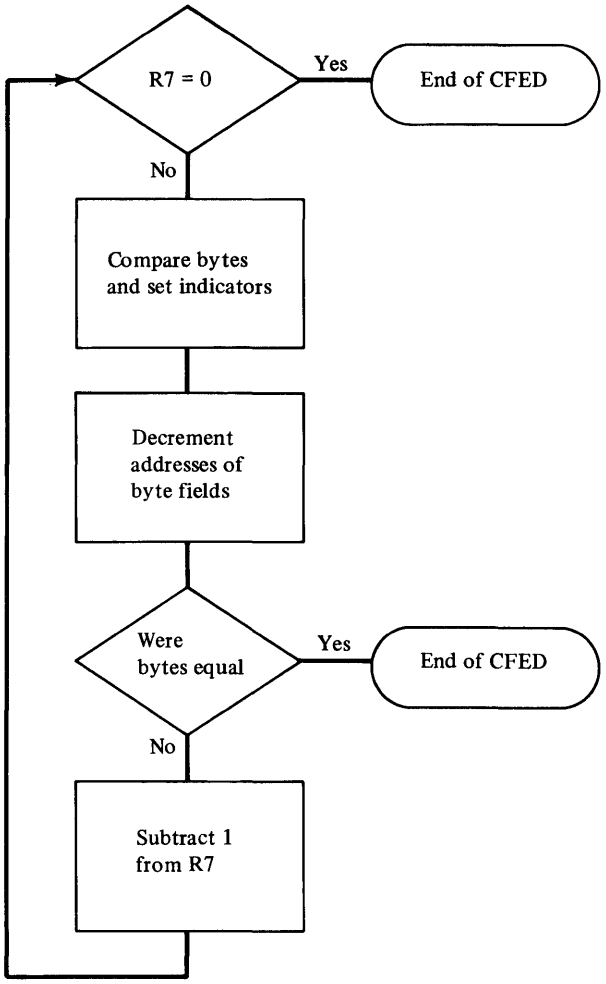
Before coding CFED, code an instruction to load register 7 with the number of bytes in each field. Both fields are the same size. (If register 7 contains 0, CFED is treated as a no-operation.)

CFED compares the rightmost byte in the second field to the rightmost byte in the first field, and sets indicators to reflect the result. The contents of both registers are decreased by 1, and now point to the bytes to the left of the ones just compared. If the two bytes were equal, CFED is finished. If they were not equal, the contents of R7 are decreased by 1, and the next bytes to the left are compared.

When CFED is finished:

- R7 contains 0 (if CFED found no *equal* condition) or the number of byte pairs not compared, plus 1 (if it found an *equal* condition).

- The first operand (*reg*) points to the byte to the left of the last byte compared in the first field.
- The second operand (*reg*) points to the byte to the left of the last byte compared in the second field.



Indicators

When CFED is finished, the indicators reflect the result of the last compare. See "Using Compare Instructions" for a description of the indicator settings.

CFED Example

```
CFED (R3),(RØ)
```

Assume that:

- R7 contains X'0003'.
- The field whose starting address is in R3 contains, in hexadecimal:
  21 21 21
  (R3 points to the rightmost byte).
- The field whose starting address is in R0 contains, in hexadecimal:
  22 23 24
  (R0 points to the rightmost byte, X'24').

CFED compares the rightmost byte in the second field, X'24', to the rightmost byte in the first field, X'21'. R0 and R3 are decreased by 1, and now point to the next bytes to the left (R0 points to X'23' and R3 points to X'21'). Because the two bytes were not equal, register 7 is decreased by 1, and CFED compares X'23' to X'21'. Because there will be no *equal* condition, CFED continues until register 7 contains 0. When CFED is finished, R0 points to the byte to the left of X'22', and R3 points to the byte to the left of the leftmost X'21'.

Compare Byte Field Equal and Increment (CFEN)

This instruction compares successive (left-to-right) bytes in one field with corresponding bytes in another field. It compares one byte pair at a time until it finds an *equal* condition OR until it has exhausted the length count.

Name	*Operation*	*Operand*
[label]	CFEN	(reg), (reg)

Code CFEN like CFED, with one exception. Load the registers with the addresses of the *leftmost* bytes in the fields.

CFEN compares the leftmost byte in the second field to the leftmost byte in the first field, and sets indicators to reflect the result. Both registers are increased by 1, and now point to the bytes to the right of the ones just compared. If the two bytes were equal, CFEN is finished. If they were not equal, register 7 is decreased by 1, and the next bytes
to the right are compared. When CFEN is finished:

- R7 contains 0 (if CFEN found no *equal* condition) or the number of byte pairs not compared, plus 1, (if it found an *equal* condition).
- The first operand (*reg*) points to the byte to the right of the last byte compared in the first field.
- The second operand (*reg*) points to the byte to the right of the last byte compared in the second field.

Indicators

When CFEN is finished, the indicators reflect the result of the last compare. See "Using Compare Instructions" for a description of the indicator settings.

CFEN Example

```
CFEN (R4),(R3)
```

Assume that:

- R7 contains X'0005'.
- The field whose starting address is in R4 contains, in hexadecimal:
  F1 F3 F5 F7 F9
  (R4 points to the leftmost byte, X'F1').
- The field whose starting address is in R3 contains, in hexadecimal:
  F1 F2 F3 F4 F5
  (R3 points to the leftmost byte, X'F1').

CFEN compares the leftmost byte in the second field, X'F1', to the leftmost byte in the first field, X'F1'. R3 and R4 are increased by 1, and now point to the next bytes to the right (R3 points to X'F2' and R4 points to X'F3'). Because the two bytes were equal, CFEN is finished. R7 contains X'0005', R4 points to X'F3' in the second field, and R3 points to X'F2' in the first field.

Compare Byte Field Not Equal and Decrement (CFNED)

This instruction compares successive (right-to-left) bytes in one field with corresponding bytes in another field. It compares one byte pair at a time until it finds a *not equal* condition OR until it has exhausted the length count.

Name	*Operation*	*Operand*
[label]	CFNED	(reg), (reg)

Code CFNED exactly like CFED.

CFNED compares the rightmost byte in the second field to the rightmost byte in the first field, and sets indicators to reflect the result. Both registers are decreased by 1, and now point to the bytes to the left of the ones just compared. If the two bytes were not equal, CFNED is finished. If they were equal, register 7 is decreased by 1, and the next bytes to the left are compared. When CFNED is finished:

- R7 contains 0 (if CFNED did not find a *not equal* condition) or the number of byte pairs not compared, plus 1 (if it found a *not equal* condition).
- The first operand (*reg*) points to the byte to the left of the last byte compared in the first field.
- The second operand (*reg*) points to the byte to the left of the last byte compared in the second field.

Indicators

When CFNED is finished, the indicators reflect the result of the last compare. See "Using Compare Instructions" for a description of the indicator settings.

CFNED Example

```
CFNED (R5),(R1)
```

Assume that:

- R7 contains X'0008'.
- The field whose starting address is in R5 contains, in hexadecimal:
  02 24 46 68 8A AC CE E0
  (R5 points to the rightmost byte, X'E0').
- The field whose starting address is in R1 contains, in hexadecimal:
  00 00 00 00 00 AC CE E0
  (R1 points to the rightmost byte, X'E0').

CFNED compares the rightmost byte in the second field, X'E0', to the rightmost byte in the first field, also X'E0'. R5 and R1 are decreased by 1, and now point to the next byte pair to the left. Because the two bytes were equal, register 7 is decreased by 1, and CFNED compares the next bytes in the field. CFNED continues until it compares X'00' (in the second field) to X'8A' (in the first field). R5 and R1 are decreased by 1, and point to the next pair to the left. Because the two compared bytes were not equal, CFNED is finished. R7 contains X'0005', R5 points to X'68' in the first field, and R1 points to X'00' in the second field.

Compare Byte Field Not Equal and Increment (CFNEN)

This instruction compares successive (left-to-right) bytes in one field with corresponding bytes in another field. It compares one byte pair at a time until it finds a *not equal* condition OR until it has exhausted the length count.

Name	*Operation*	*Operand*
[label]	CFNEN	(reg), (reg)

Code CFNEN exactly like CFEN.

CFNEN compares the leftmost byte in the second field to the leftmost byte in the first field, and sets indicators to reflect the result. Both registers are increased by 1, and now point to the bytes to the right of the ones just compared. If the two bytes were not equal, CFNEN is finished. If they were equal, register 7 is decreased by 1, and the next bytes to the right are compared. When CFNEN is finished:

- R7 contains 0 (if CFNEN did not find a *not equal* condition) or the number of byte pairs not compared, plus 1 (if it found a *not equal* condition).
- The first operand (*reg*) points to the byte to the right of the last byte compared in the first field.
- The second operand (*reg*) points to the byte to the right of the last byte compared in the second field.

Indicators

When CFNEN is finished, the indicators reflect the result of the last compare. See "Using Compare Instructions" for a description of the indicator settings.

CFNEN Example

```
CFNEN (R4),(R1)
```

Assume that:

- R7 contains X'000D'.
- The field whose starting address is in R4 contains, in hexadecimal:
  FF FF FF 12 12 12 12 12 12 12 12 12 12
  (R4 points to the leftmost byte, X'FF').
- The field whose starting address is in R1 contains, in hexadecimal:
  12 FF FF FF FF FF FF FF FF FF FF FF FF
  (R1 points to the leftmost byte, X'12').

CFNEN compares the leftmost byte in the second field, X'12', to the leftmost byte in the first field, X'FF'. R4 and R1 are increased by 1, and now point to the next bytes to the right. Because the two bytes were not equal, CFNEN is finished. R7 contains X'000D', R1 and R4 point to the bytes to the right of the ones just compared.

Compare Byte Immediate (CBI)

This instruction compares a specified register to a 1-byte absolute value or expression, and sets indicators to reflect the result.

Name	*Operation*	*Operand*
[label]	CBI	byte, reg

For *byte*, code an 8-bit value in the range −128 to 127 (arithmetic) or 0 to 255 (logical). CBI expands this value to 16 bits by propagating the sign bit to the left of the *byte* value. This value is then used in the comparison.

Note. A logical value in the range 128 to 255 will propagate a sign bit of one to the left.

Indicators

See "Using Compare Instructions" for a description of the indicator settings.

CBI Example

```
CBI 125,R7
```

Assume that R7 contains X'02A6'. CBI expands X'7D' (the equivalent of decimal 125) to 16 bits by propagating the sign bit (zero) to the left. CBI then compares X'02A6' to X'007D', and sets *arithmetically greater than* and *logically greater than* conditions.

Compare Doubleword (CD)

This instruction compares:

- A doubleword in a register pair to a doubleword in storage, or
- A doubleword in storage to a doubleword in storage,

and sets indicators to reflect the result.

Name	*Operation*	*Operand*
[label]	CD	addr4, reg addr5, addr4

Indicators

See "Using Compare Instructions" for a description of the indicator settings.

CD Example

```
CD DWORD,R2
```

Assume that:

- DWORD contains X'00E4E1C0', and
- The register pair R2,R3 contains X'018F3A66'.

CD compares the two values, and sets *arithmetically greater than* and *logically greater than* conditions.

Compare Word (CW)

This instruction compares:

- A word in a register to a word in a register,
- A word in a register to a word in storage, or
- A word in storage to a word in storage,

and sets indicators to reflect the result.

Name	*Operation*	*Operand*
[label]	CW	reg, reg addr4, reg addr5, addr4

Indicators

See "Using Compare Instructions" for a description of the indicator settings.

CW Example

```
CW WORD,R3
```

Assume that WORD contains X'369C' and register 3 contains X'0D02'. CW compares the two values, and sets *arithmetically less than* and *logically less than* conditions.

Compare Word Immediate (CWI)

This instruction compares:

- A word in a register to a 1-word absolute value or expression, or
- A word in storage to a 1-word absolute value or expression,

and sets indicators to reflect the result.

Name	*Operation*	*Operand*
[label]	CWI	word, reg word, addr4

Indicators

See "Using Compare Instructions" for a description of the indicator settings.

CWI Example

```
CWI -766,RØ
```

The immediate word value is equal to X'FD02'. Assume that register 0 contains X'369C'. CWI compares the values, and sets *arithmetically greater than* and *logically less than* conditions.

Scan Byte Field Equal and Decrement (SFED)

This instruction compares successive bytes in a field (right-to-left) to a byte in a register until it either finds an *equal* condition or exhausts the length count.

Name	*Operation*	*Operand*
[label]	SFED	reg, (reg)

Before coding SFED, code instructions to:

- Load register 7 with the number of bytes in the field to be scanned.
- Load *reg* so that bits 8–15 of the register contain the byte that the field is to be scanned for.
- Load the (*reg*) register with the address of the rightmost byte in the field to be scanned.

SFED compares the rightmost byte in the field to the *reg* byte, and sets indicators to reflect the result. The (*reg*) register is decreased by 1, and now points to the byte to the left of the one just compared. If the two bytes were equal, SFED is finished. If they were not equal, register 7 is decreased by 1, and the next byte is compared.

Indicators

When SFED is finished, the indicators reflect the result of the last compare. See "Using Compare Instructions" for a description of the indicator settings.

SFED Example

```
SFED R2,(R4)
```

Assume that:

- Register 7 contains X'0003', the size of the byte field to be scanned.
- Bits 8–15 of register 2 contain X'6D', the byte to be compared to the field.
- The contents of the byte field defined by register 4 are, in hexadecimal:
  A4 6D 53
  where R4 points to the rightmost byte in the field (X'53').

SFED compares X'53' to X'6D', then decreases R4 by 1. Because the two bytes were not equal, SFED decreases R7 by 1. Now, R4 points to the next byte to the left of the one just compared, and R7 contains X'0002', the number of bytes yet to be compared. SFED compares the byte in the field (X'6D') to the R2 byte (X'6D'), and again decreases R4 by 1. Because the two bytes were equal, SFED is done. R4 points to X'A4', and R7 contains X'0002'.

Scan Byte Field Equal and Increment (SFEN)

This instruction compares successive bytes in a field (left-to-right) to a byte in a register until it either finds an *equal* condition or exhausts the length count.

Name	*Operation*	*Operand*
[label]	SFEN	reg, (reg)

Before coding SFEN, code instructions to:

- Load register 7 with the number of bytes in the field.
- Load *reg* so that bits 8–15 of the register contain the byte that the field is to be scanned for.
- Load the (*reg*) register with the address of the leftmost byte in the field.

SFEN compares the leftmost byte in the field to the *reg* byte, and sets indicators to reflect the result. The (*reg*) register is increased by 1, and now points to the byte to the right of the one just compared. If the two bytes were equal, SFEN is finished. If they were not equal, register 7 is decreased by 1, and the next byte is compared.

Indicators

When SFEN is finished, the indicators reflect the result of the last compare. See "Using Compare Instructions" for a description of the indicator settings.

SFEN Example

```
SFEN R5,(R1)
```

Assume that:

- Register 7 contains X'0005', the size of the byte field to be scanned.
- Bits 8–15 of register 5 contain X'00', the byte to be compared to the field.
- The contents of the byte field defined by register 1 are, in hexadecimal:
  10 83 B5 4A FF
  where R1 points to the leftmost byte in the field (10).

SFEN compares X'10' to X'00', then increases R1 by 1. Because the two bytes were not equal, SFEN decreases R7 by 1. Now, R1 points to the next byte to the right of the one just compared, and R7 contains X'0004', the number of bytes yet to be compared. Because no byte in the field is equal to the byte in R5, SFEN compares until it exhausts the field. When SFEN is finished, R1 points to the byte to the right of X'FF'. R7 contains X'0000'.

Scan Byte Field Not Equal and Decrement (SFNED)

This instruction compares successive bytes in a field (right-to-left) to a byte in a register until it either finds a *not equal* condition or exhausts the length count.

Name	*Operation*	*Operand*
[label]	SFNED	reg, (reg)

Code SFNED exactly like SFED.

SFNED compares the rightmost byte in the field to the *reg* byte, and sets indicators to reflect the result. The (*reg*) register is decreased by 1, and now points to the byte to the left of the one just compared. If the two bytes were not equal, SFNED is finished. If they were equal, register 7 is decreased by 1, and the next byte is compared.

Indicators

When SFNED is finished, the indicators reflect the result of the last compare. See "Using Compare Instructions" for a description of the indicator settings.

SFNED Example

```
SFNED R3,(R6)
```

Assume that:

- Register 7 contains X'0009', the size of the byte field to be scanned.
- Bits 8–15 of register 3 contain X'FF', the byte to be compared to the field.
- The contents of the field defined by register 6 are, in hexadecimal:
  FF FF 00 FF FF FF FF FF FF where R6 points to the rightmost byte in the field.

SFNED compares the rightmost byte in the field to the *reg* byte, then decreases R6 by 1. Because the two bytes were equal, SFNED decreases R7 by 1, and compares the next byte to the left. SFNED continues until it reaches the byte X'00'. It compares the *reg* byte to X'00', decreases R6 by 1, and, because the two bytes were unequal, SFNED is finished. R7 contains X'0003'.

Scan Byte Field Not Equal and Increment (SFNEN)

This instruction compares successive bytes in a field (left-to-right) with a byte in a register until it either finds a *not equal* condition or exhausts the length count.

Name	*Operation*	*Operand*
[label]	SFNEN	reg, (reg)

Code SFNEN exactly like SFEN.

SFNEN compares the leftmost byte in the field to the *reg* byte, and sets indicators to reflect the result. The (*reg*) register is increased by 1, and now points to the byte to the right of the one just compared. If the two bytes were not equal, SFNEN is finished. If they were equal, register 7 is decreased by 1, and the next byte is compared.

Indicators

When SFNEN is finished, the indicators reflect the result of the last compare. See "Using Compare Instructions" for a description of the indicator settings.

SFNEN Example

```
SFNEN RØ,(R1)
```

Assume that:

- Register 7 contains X'0004'.
- Bits 8–15 of R0 contain X'00'.
- The byte field defined by R1 contains:
  00 00 00 00
  where R1 points to the leftmost byte.

Because the *reg* byte is equal to every byte in the field, SFNEN compares until R7 contains X'0000'. When SFNEN is finished, R1 points to the right of the last byte compared.

Logical Instructions

AND Word Immediate (NWI)

This instruction performs an AND operation between an immediate word value and a register.

Name	*Operation*	*Operand*
[label]	NWI	word, reg[,reg]

Note the optional third operand. If you code this third operand, NWI places its result into that register, leaving the second operand unchanged. Otherwise, the result is placed in the register specified by the second operand. In either case, the *word* operand remains unchanged.

Indicators

The carry and overflow indicators are unchanged. The remaining indicators reflect the result.

NWI Example

```
NWI X'13A1',R6,R3
```

In this example, the immediate value looks like this:
0001 0011 1010 0001
Assume that register 6 contains:
0101 1101 1110 0111
The result that NWI places in register 3 is:
0001 0001 1010 0001

Exclusive OR Byte (XB)

This instruction performs an exclusive OR between a byte in a register and a byte in storage.

Name	*Operation*	*Operand*
[label]	XB	reg, addr4 addr4, reg

If you code the *reg,addr4* form, XB exclusive ORs bits 8–15 of *reg* and the byte at *addr4*, placing the result at *addr4*. The register is unchanged.

In the *addr4,reg* form, XB places the result in bits 8–15 of *reg*, leaving bits 0–7 unchanged. The byte at *addr4* is not altered.

Indicators

The carry and overflow indicators are unchanged. The remaining indicators reflect the 8-bit result.

XB Example

```
XB BYTE6,R3
```

Assume that the byte at BYTE6 contains:
0001 1101
and bits 8–15 of register 3 contain:

0110 0110
The result that XB places in bits 8–15 of R3 is:
0111 1011

Exclusive OR Doubleword (XD)

This instruction performs an exclusive OR between a doubleword in a register pair and a doubleword in storage.

Name	*Operation*	*Operand*
[label]	XD	reg, addr4 addr4, reg

For the *reg* operand, code the first register of a pair. If you code R3, for example, XD uses the register pair R3,R4. If you code R7, XD uses the pair R7,R0.

XD exclusive ORs the first operand to the second, and places the result in the second operand. The first operand remains unchanged.

Indicators

The carry and overflow indicators are unchanged. The remaining indicators reflect the 32-bit result.

XD Example

```
XD DWORD,R7
```

Assume that the doubleword at DWORD contains:
0111 1011 0001 1101 0110 0110 0111 1011
and the register pair R7,R0 contains:
0001 1101 0110 0110 0111 1011 1011 1100
The result that XD places in R7,R0 is:
0110 0110 0111 1011 0001 1101 1100 0111

Exclusive OR Word (XW)

This instruction performs an exclusive OR between:

- A register and a register, or

Name	*Operation*	*Operand*
[label]	XW	reg, reg reg, addr4 addr4, reg longaddr, reg

XW exclusive ORs the first operand to the second, placing the result in the second operand. The first operand remains unchanged.

Indicators

The carry and overflow indicators are unchanged. The remaining indicators reflect the 16-bit result.

XW Example

```
XW (R6)*,RØ
```

In this example (coded in *longaddr,reg* form), the first operand is the word whose address is the contents of storage at the location defined by register 6. Assume that this word contains:
0110 0111 0001 1100
and register 0 contains:
0111 1011 0001 1101
The result that XW places in R0 is:
0001 1100 0000 0001

Exclusive OR Word Immediate (XWI)

This instruction performs an exclusive OR between a 1-word absolute expression and a register.

Name	*Operation*	*Operand*
[label]	XWI	word, reg[,reg]

Note that there is an optional third operand. If you code it, XWI places the result in this register, leaving the second operand unchanged. Otherwise, the result is placed in the register defined by the second operand. In either case, the *word* operand remains unchanged.

Indicators

The carry and overflow indicators are unchanged. The remaining indicators reflect the 16-bit result.

XWI Example

```
XWI X'Ø6BD',R1,R5
```

The immediate value, X'06BD', looks like this:
0000 0110 1011 1101
Assume that R1 contains:
0110 1001 1000 0001
The value that XWI places in R5 is:
0110 1111 0011 1100

Invert Register (VR)

This instruction produces the ones complement of the contents of the specified register.

Name	*Operation*	*Operand*
[label]	VR	reg[,reg]

Note the optional second operand. If you code this register, VR places the result (in ones complement form) in that register, leaving the first operand unchanged. If you don't code the second operand, VR places the complement back into the source register.

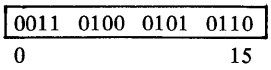
Indicators

The carry indicator and overflow indicators are unchanged. The other indicators are changed to reflect the result.

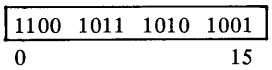
VR Example

```
VR R3,R4
```

Assume that register 3 contains:



After execution of VR, register 4 contains:



Register 3 is unchanged.

OR Byte (OB)

This instruction performs an OR operation between:

- A byte in a register and a byte in storage, or
- A byte in storage and another byte in storage.

Name	*Operation*	*Operand*
[label]	OB	reg, addr4 addr4, reg addr5, addr4

If you code the *reg,addr4* form, OB ORs bits 8–15 of *reg* and the byte at *addr4*, placing the result at *addr4*. The register is unchanged.

In the *addr4,reg* form, OB places the result in bits 8–15 of reg, leaving bits 0–7 unchanged. The byte at *addr4* is not altered.

If you code *addr5,addr4*, the result is placed at *addr4*, leaving *addr5* unchanged.

Indicators

The carry and overflow indicators are unchanged. The remaining indicators reflect the 8-bit result.

OB Example

```
OB (R5)*,(R2)*
```

In this example (coded in *addr5,addr4* form), the first operand is the byte in storage whose address is the contents of storage at the location defined by register 5. Assume that this byte contains:
0111 0001
The second operand is the byte in storage whose address is the contents of storage at the location defined by register 2. Assume that this byte contains:
0001 0100
The result that OB places in the byte specified by the second operand is:
0111 0101

OR Doubleword (OD)

This instruction performs an OR operation between:

- A doubleword in a register pair and a doubleword in storage, or
- A doubleword in storage and another doubleword in storage.

Name	*Operation*	*Operand*
[label]	OD	reg, addr4 addr4, reg addr5, addr4

For the *reg* operand, code the first register of a pair. If you code R3, for example, OD uses registers 3 and 4. If you code R7, OD uses the pair R7,R0.

OD ORs the first operand to the second, and places the result in the second operand. The first operand remains unchanged.

Indicators

The carry and overflow indicators are unchanged. The remaining indicators reflect the 32-bit result.

OD Example

```
OD (R7),(R1)
```

In this example (coded in *addr5,addr4* form), the first operand is the doubleword in storage whose address is in register 7. Assume that this doubleword contains:
0110 0110 0111 1011 0001 1101 1100 0111
The second operand is the doubleword in storage whose address is the contents of register 1. Assume that this doubleword contains:
0111 1011 0001 1101 0110 0110 0111 1011
The result that OD places in the doubleword defined by the second operand is:
0111 1111 0111 1111 0111 1111 1111 1111

OR Word (OW)

This instruction performs an OR operation between:

- A register and a register,
- A register and a word in storage, or
- A word in storage and another word in storage.

Name	*Operation*	*Operand*
[label]	OW	reg, reg reg, addr4 addr4, reg longaddr, reg addr5, addr4

OW ORs the first operand to the second, and places the result in the second operand. The first operand remains unchanged.

Indicators

The carry and overflow indicators are unchanged. The remaining indicators reflect the result.

OW Example

```
OW (R5,3),R4
```

In this example (coded in *longaddr,reg* form), the first operand is the word in storage that is 3 bytes past the address specified by register 5. Assume that this word contains:

0111 1011 0001 1101
and register 4 contains:
0001 1100 0000 0001
The result that OW places in register 4 is:
0111 1111 0001 1101

OR Word Immediate (OWI)

This instruction performs an OR operation between:

- A 1-word absolute expression and a register, or
- A 1-word absolute expression and a word in storage.

Name	*Operation*	*Operand*
[label]	OWI	word, reg[,reg] word, addr4

Note the optional third operand in the *word,reg,[reg]* form. If you code this third operand, the result of the OR is placed in the register you code, leaving the second operand unchanged. Otherwise, OWI places the result in the register specified for the second operand. In either case, the *word* operand remains unchanged.

If you code the *word,addr4* form, the result is placed in *addr4*, leaving the *word* operand unchanged.

Note. The *word* operand is an absolute value or expression in the range −32768 to +32767 or 0 to 65535.

Indicators

The carry and overflow indicators are unchanged. The remaining indicators reflect the result.

OWI Example

```
OWI X'13A1',WORD
```

In this example (coded in *word,addr4* form), the immediate value looks like this:
0001 0011 1010 0001
Assume that the word at storage location WORD is:
0100 0101 0110 0111
The result that OWI places in WORD is:
0101 0111 1110 0111

Reset Bits Byte (RBTB)

This instruction operates on a byte, setting specified bits to zero.

Name	*Operation*	*Operand*
[label]	RBTB	reg, addr4 addr4, reg addr5, addr4

RBTB finds the bits that are *on* in the byte defined by the first operand. It then turns *off* the corresponding bits in the byte defined by the second operand. The first operand is unchanged.

If you code the *reg,addr4* form or the *addr4,reg* form, RBTB uses bits 8–15 of *reg*, leaving bits 0–7 unchanged.

Indicators

The carry and overflow indicators are unchanged. The remaining indicators reflect the 8-bit result.

RBTB Example

```
RBTB R3,BYTES
```

In this example (coded in *reg,addr4* form), assume that bits 8–15 of R3 contain:
0111 0010
and the byte at BYTES contains:
0010 1111
Because the 1, 2, 3, and 6 bits of the R3 byte are *on*, RBTB turns *off* the corresponding bits in BYTES:
0000 1101

Reset Bits Doubleword (RBTD)

This instruction operates on a doubleword, setting specified bits to zero.

Name	*Operation*	*Operand*
[label]	RBTD	reg, addr4 addr4, reg addr5, addr4

RBTD finds the bits that are *on* in the doubleword defined by the first operand. It then turns *off* the corresponding bits in the doubleword defined by the second operand. The first operand is unchanged.

For *reg*, code the first register of a pair. For example, if you code R5, RBTD uses registers 5 and 6. If you code R7, RBTD uses the pair R7,R0.

Indicators

The carry and overflow indicators are unchanged. The remaining indicators reflect the result.

RBTD Example

```
RBTD 4(R3,48)*,R6
```

In this example (coded in *addr4,reg* form), the location of the doubleword defined by *addr4* is computed as follows: The contents of R3, added to 48, form an address. The contents of storage at that location are added to 4, forming the address of the doubleword. Assume that this doubleword contains:
0011 0101 0110 0111 1111 1110 1101 1100
and the register pair R6,R7 contains:
0000 0010 0100 0110 1000 1100 1110 1111
RBTD finds the bits that are *on* in the first operand, and sets the corresponding bits *off* in the second operand. The result that RBTD leaves in R6,R7 is:
0000 0010 0000 0000 0000 0000 0010 0011

Reset Bits Word (RBTW)

This instruction operates on a word, setting specified bits to zero.

Name	*Operation*	*Operand*
[label]	RBTW	reg, reg reg, addr4 addr4, reg longaddr, reg addr5, addr4

RBTW finds the bits that are *on* in the word defined by the first operand. It then turns *off* the corresponding bits in the word defined by the second operand. The first operand is unchanged.

Indicators

The carry and overflow indicators are unchanged. The remaining indicators reflect the result.

RBTW Example

```
RBTW (R6,36)*,(R2)+
```

In this example (coded in *addr5,addr4* form), the location of the word defined by *addr5* is computed as follows: The contents of R6, plus 36, form an address. The contents of storage at that location are the address of the word. Assume that this word contains:
0000 0100 0101 0111
The second word is at the address defined by the contents of R2. (After RBTW, R2 is increased by 2, the number of bytes addressed by this instruction.) Assume that this word contains:
1100 1111 1011 0001
The result that RBTW leaves in the second operand is:
1100 1011 1010 0000

Reset Bits Word Immediate (RBTWI)

This instruction operates on a word, setting to zero the bits specified by an immediate value.

Name	*Operation*	*Operand*
[label]	RBTWI	word, addr4 word, reg[,reg]

RBTWI finds the bits that are *on* in the immediate value defined by the *word* operand. It then turns *off* the corresponding bits in the word defined by the second operand. The first operand remains unchanged.

Note the optional third operand in the *word,reg[,reg]* form. If you code this register, RBTWI places the result there, leaving the second operand unchanged. If you do not code the third operand, the result is placed in the second operand.

Indicators

The carry and overflow indicators are unchanged. The remaining indicators reflect the result.

RBTWI Example

```
RBTWI X'4567',RØ,R1
```

In this example (coded in *word,reg[,reg]* form), the immediate word value looks like this:
0100 0101 0110 0111
Assume that R0 contains:
0111 0110 0101 0100
The result that RBTWI places in R1 is:
0011 0010 0001 0000
The immediate value and R0 are unchanged.

Set Bits Byte (SBTB)

This instruction operates on a byte, setting specified bits to one.

Name	*Operation*	*Operand*
[label]	SBTB	reg, addr4 addr4, reg addr5, addr4

SBTB finds the bits that are *on* in the byte defined by the first operand. It then turns *on* the corresponding bits in the byte defined by the second operand. The first operand is unchanged.

If you code the reg,addr4 form or the addr4,reg form, SBTB uses bits 8–15 of reg, leaving bits 0–7 unchanged.

Indicators

The carry and overflow indicators are unchanged. The remaining indicators reflect the 8-bit result.

SBTB Example

```
SBTB      SET,R7
```

In this example, assume that the SET byte contains:
0001 1000
Assume that bits 8–15 of R7 contain:
0010 0110
The result that SBTB places in bits 8–15 of R7 is:
0011 1110
The SET byte is unchanged, as are bits 0–7 of R7.

Set Bits Doubleword (SBTD)

This instruction operates on a doubleword, setting specified bits to one.

Name	*Operation*	*Operand*
[label]	SBTD	reg, addr4 addr4, reg addr5, addr4

SBTD finds the bits that are *on* in the doubleword defined by the first operand. It then turns *on* the corresponding bits in the doubleword specified by the second operand. The first operand is unchanged.

For reg, code the first register of a pair. For example, if you code R5, SBTD uses registers 5 and 6. If you code R7, SBTD uses the pair R7,R0.

Indicators

The carry and overflow indicators are unchanged. The remaining indicators reflect the 32-bit result.

SBTD Example

```
SBTD      R0,(R2)
```

In this example, assume that the register pair R0,R1 contains:
0000 0110 0001 1101 0010 1010 0100 1111
and the doubleword defined by the address in R2 contains:
0111 0000 0000 0000 0000 0000 0000 1111
The result that SBTD places in the doubleword specified by the second operand is:
0111 0110 0001 1101 0010 1010 0100 1111
Registers 0, 1, and 2 are unchanged.

Set Bits Word (SBTW)

This instruction operates on a word, setting specified bits to one.

Name	*Operation*	*Operand*
[label]	SBTW	reg, reg reg, addr4 addr4, reg longaddr, reg addr5, addr4

SBTW finds the bits that are *on* in the word defined by the first operand. It then turns *on* the corresponding bits in the word specified by the second operand. The first operand is unchanged.

Indicators

The carry and overflow indicators are unchanged. The remaining indicators reflect the 16-bit result.

SBTW Example

```
SBTW      R5,R2
```

In this example, assume that R5 contains:
0001 0001 0000 0000
and R2 contains:
0001 0010 0011 0100
The value that SBTW places in R2 is:
0001 0011 0011 0100
Register 5 is unchanged.

Set Bits Word Immediate (SBTWI)

This instruction operates on a word, setting to one the bits specified by an immediate operand.

Name	*Operation*	*Operand*
[label]	SBTWI	word, reg [,reg] word, addr4

SBTWI finds the bits that are *on* in the immediate value defined by the word operand. It then turns *on* the corresponding bits in the word defined by the second operand. The first operand remains unchanged.

Note the optional third operand in the word,reg[,reg] form. If you code this register, SBTWI places the result there, leaving the second operand unchanged. If you do not code the third operand, the result is placed in the second operand.

Indicators

The carry and overflow indicators are unchanged. The remaining indicators reflect the result.

SBTWI Example

```
SBTWI     X'1234',R3,R0
```

In this example (coded in word,reg[,reg] form), the immediate word value looks like this:
0001 0010 0011 0100
Assume that R3 contains:
0111 1011 0001 1100
The result that SBTWI places in R0 is:
0111 1011 0011 1100
The immediate value and R3 are unchanged.

Test Bit (TBT)

This instruction tests a single bit, and sets an indicator to reflect the result.

Name	*Operation*	*Operand*
[label]	TBT.	(reg, bitdisp)

To find the bit to be tested, TBT uses the contents of *reg* as a byte address, and the value of *bitdisp* as a displacement from the byte address.

Note. *Bitdisp* must be in the range 0–63.

Here is what TBT does:

- It turns off the zero and negative indicators, then
- It tests the specified bit.
- If the bit is zero, TBT turns on the zero indicator; if the bit is one, TBT turns on the negative indicator.

Indicators

The zero and negative indicators reflect the result of the test. The remaining indicators are unchanged.

TBT Example

```
TBT (R7,3)
```

Assume that:

- R7 contains X'0420'.
- The byte at address 0420 contains:
  0000 1111

TBT turns off the zero and negative indicators, then tests bit number 3 (the fourth bit in the byte). Because the bit is zero, TBT turns on the zero indicator.

Test Bit and Invert (TBTV)

This instruction tests a single bit, and sets an indicator to reflect the result. After setting the appropriate indicator, TBTV unconditionally inverts the tested bit.

Name	*Operation*	*Operand*
[label]	TBTV	(reg, bitdisp)

TBTV computes the address of the bit in the same way as TBT, and follows the same procedure for testing the bit and setting the indicator.

Indicators

The zero and negative indicators reflect the result of the test. The remaining indicators are unchanged.

TBTV Example

```
TBTV (R7,3)
```

Assume that:

- R7 contains X'0420'.
- The byte at address 0420 contains:
  0000 1111

TBTV turns off the zero and negative indicators, then tests bit number 3 (the fourth bit in the byte). Because the bit is zero, TBTV turns on the zero indicator. TBTV inverts the tested bit, and the byte now looks like this:
0001 1111

Test Bit and Reset (TBTR)

This instruction tests a single bit, and sets an indicator to reflect the result. After setting the appropriate indicator, TBTR unconditionally sets the tested bit to zero.

Name	*Operation*	*Operand*
[label]	TBTR	(reg, bitdisp)

TBTR computes the address of the bit in the same way as TBT, and follows the same procedure for testing the bit and setting the indicator.

Indicators

The zero and negative indicators reflect the result of the test. The remaining indicators are unchanged.

TBTR Example

```
TBTR (R2,Ø)
```

Assume that:

- R2 contains X'0348'.
- The byte at address 0348 contains:
  1010 1110

TBTR turns off the zero and negative indicators, then tests bit number 0 (the first bit in the byte). Because the bit is one, TBTR turns on the negative indicator. TBTR sets the tested bit to zero, and the byte now looks like this:
0010 1110

Test Bit and Set (TBTS)

This instruction tests a single bit, and sets an indicator to reflect the result. After setting the appropriate indicator, TBTS unconditionally sets the tested bit to one.

Name	*Operation*	*Operand*
[label]	TBTS	(reg, bitdisp)

TBTS computes the address of the bit in the same way as TBT, and follows the same procedure for testing the bit and setting the indicator.

Indicators

The zero and negative indicators reflect the result of the test. The remaining indicators are unchanged.

TBTS Example

```
TBTS (R3,5)
```

Assume that:

- R3 contains X'8680'.
- The byte at address 8680 contains:
  1101 0010

TBTS turns off the zero and negative indicators, then tests bit number 5 (the sixth bit in the byte). Because the bit is zero, TBTS turns on the zero indicator. TBTS sets the tested bit to one, and the byte now looks like this:
1101 0110

Test Word Immediate (TWI)

This instruction tests specified bits within a word, and sets an indicator to reflect the result.

Name	*Operation*	*Operand*
[label]	TWI	word, reg word, addr4

For the *word* operand, code a 1-word mask. TWI finds the bits that are on in the mask and tests the corresponding bits in the word defined by the second operand. TWI clears the zero and negative result indicators, then sets them as follows:

- If the mask bits or all of the tested bits are zeros, TWI turns on the zero indicator.
- If all the tested bits are ones, TWI turns on the negative indicator.
- If the tested bits are a combination of zeros and ones, TWI sets no indicators (sets a *positive* condition).

Indicators

The even, carry, and overflow indicators are unchanged. The remaining indicators reflect the result.

TWI Example

```
TWI X'13A1',R4
```

The word mask looks like this:
0001 0011 1010 0001
Assume that R4 contains:
1011 0111 1110 0001
Because all the tested bits are ones, TWI turns on the negative indicator.

Processor Status Instructions

Copy Level Status Register (CPLSR)

This instruction loads the contents of the current Level Status Register (LSR) into a specified register.

Name	*Operation*	*Operand*
[label]	CPLSR	reg

Note. For information about the contents of the LSR see Chapter 3 of this manual.

Indicators

All indicators are unchanged.

CPLSR Example

```
CPLSR R5
```

The contents of the LSR are placed in R5, and the LSR remains unchanged.

Set Indicators (SEIND)

This instruction stores the contents of bits 0–4 of a specified register into the result indicators in the Level Status Register (bits 0–4).

Name	*Operation*	*Operand*
[label]	SEIND	reg

SEIND stores bits 0–4 of *reg* into the even, carry, overflow, negative, and zero indicators. Bits 5–15 of the Level Status Register are unchanged. Bits 5–15 of the register are ignored.

Indicators

The indicators contain the values specified by bits 0–4 of *reg*.

SEIND Example

```
SEIND R1
```

Assume that register 1 contains:
1101 0000 1101 0110
The result that SEIND places into bits 0–4 of the Level Status Register is:

Bit 0 even=1
Bit 1 carry=1
Bit 2 overflow=0
Bit 3 negative=1
Bit 4 zero=0

Stop (STOP)

This instruction causes the processor to enter the stop state.

Name	*Operation*	*Operand*
[label]	STOP	[ubyte]

For the *ubyte* operand, you can optionally code a 1-byte unsigned absolute value or expression. The processor ignores this value, so you can use it for a flag or indicator.

Note. *ubyte* defaults to zero when it's not coded.

For STOP to stop the processor, the processor must have a full-function console, and the auto-IPL switch must be in the "Diagnostic Mode" position. Otherwise, STOP is executed as a no-operation instruction, causing control to be passed to the next sequential instruction.

Indicators

All indicators are unchanged.

Supervisor Call (SVC)

This instruction interrupts the program being executed, then passes control to the supervisor so it can perform the service specified by the operand.

Name	*Operation*	*Operand*
[label]	SVC	[ubyte]

For the *ubyte* operand, you can optionally code a 1-byte unsigned absolute value or expression. The value is loaded into the low-order byte of R1. The high-order byte of R1 is set to zero. Control is passed to the address that is in location X'0012'. For more information about supervisor state, refer to the processor description manual for your processor.

Indicators

All indicators are unchanged.

Privileged Instructions

Copy Address Key Register (CPAKR) (4955 Processor Only)

This instruction copies the value in bits 0–15 from the Address Key Register(AKR) into a storage location or a register.

Name	*Operation*	*Operand*
[label]	CPAKR	addr4 reg

If you code the *addr4* operand, CPAKR copies the contents of the AKR into bits 0–15 of this word.

If you code the *reg* operand, the contents of the AKR are copied into bits 0–15 of the specified register.

Indicators

All indicators are unchanged.

CPAKR Example

```
CPAKR R6
```

The contents of the AKR are stored in R6, and the AKR remains unchanged.

Copy Console Data Buffer (CPCON)

This instruction places the contents of the console data buffer into a specified register.

Name	*Operation*	*Operand*
[label]	CPCON	reg

Notes.

1. If your processor does not have the full-function console, the contents of the register are undefined.
2. For information about the console data buffer, refer to the processor description manual for your processor. See the Preface of this manual for titles and order numbers.

Indicators

All indicators are unchanged.

CPCON Example

```
CPCON R5
```

The contents of the console data buffer are placed in R5, and the console data buffer remains unchanged.

Copy Current Level (CPCL)

The Copy Current Level (CPCL) instruction loads the current level into the specified register.

Name	*Operation*	*Operand*
[label]	CPCL	reg

Indicators

All indicators are unchanged.

CPCL Example

```
CPCL R3
```

Assume that your program is currently running on level 1. The value that CPCL places in register 3 is X'0001'.

Copy In-Process Flags (CPIPF)

This instruction places the value of the in-process flag for each level (bit 9 of each Level Status Register) into a word in storage.

Name	*Operation*	*Operand*
[label]	CPIPF	addr4

For *addr4*, code the address of the word in storage where the in-process flags are to be stored. Each bit in the word then corresponds to an interrupt level's in-process flag. For example, bit 0 of the word corresponds to level 0—if the in-process flag for level 0 is on, CPIPF places a 1 in bit 0 of *addr4*; if the flag is off, CPIPF places a 0 in bit 0 of *addr4*. Bit 1 of *addr4* corresponds to level 1, bit 2 corresponds to level 2, and so on.

Bits corresponding to nonexistent levels are set to zero, and the in-process flags remain unchanged.

Indicators

All indicators are unchanged.

CPIPF Example

```
CPIPF (R1)
```

Assume that the in-process flags for each level are:

Level 0 Flag=0
Level 1 Flag=0
Level 2 Flag=1
Level 3 Flag=0

The result that CPIPF leaves in the word (whose address is in register 1) is:
0010 0000 0000 0000

Copy Instruction Space Key (CPISK) (4955 Processor Only)

This instruction copies the instruction space key field (bits 13–15) within the Address Key Register (AKR) into bits 13–15 of either a word in storage or a register. Bits 0–12 are set to zero.

Name	*Operation*	*Operand*
[label]	CPISK	addr4 reg

If you code the *addr4* operand, CPISK copies the contents of bits 13–15 from the AKR into bits 13–15 of a word in storage defined by *addr4*.

If you code the *reg* operand, CPISK copies the contents of bits 13–15 of the AKR into bits 13–15 of the specified register.

Indicators

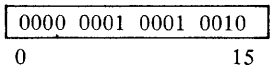
All indicators are unchanged.

CPISK Example

```
CPISK R4
```

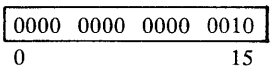
Assume the AKR contains X'0112':

AKR



After execution of CPISK register 4 contains:

R4



The instruction space key field is copied into bits 13–15, and bits 0–12 are set to zero. The AKR remains unchanged.

Copy Interrupt Mask Register (CPIMR)

This instruction places the contents of the Interrupt Mask Register into a word in storage.

Name	*Operation*	*Operand*
[label]	CPIMR	addr4

Each bit of the Interrupt Mask Register corresponds to an interrupt level; for example, bit 3 corresponds to level 3. Bits corresponding to nonexistent levels are always zero. CPIMR leaves the Interrupt Mask Register unchanged.

Indicators

All indicators are unchanged.

CPIMR Example

```
CPIMR (R3)*
```

In this example, CPIMR places the contents of the Interrupt Mask Register into the word whose address is the contents of storage at the location defined by register 3.

Copy Level Status Block (CPLB)

This instruction places the contents of a specified Level Status Block into a word-aligned 22-byte storage area that you define.

Name	*Operation*	*Operand*
[label]	CPLB	reg, addr4

The *reg* operand defines the interrupt level whose Level Status Block is to be copied. Load this register so that the level is specified in bits 12–15, with bits 0–11 containing zeros. CPLB ignores bits corresponding to nonexistent levels.

The *addr4* operand defines the first word of a 22-byte (11-word) storage area where the Level Status Block is to be stored. CPLB stores the Level Status Block in the following format:

Main storage address (LSB pointer)

Instruction address register
Address key register *
Level status register
Register 0
Register 1
Register 2
Register 3
Register 4
Register 5
Register 6
Register 7

0 15

*4955 Processor *only*

The Level Status Block remains unchanged.

Indicators

All indicators are unchanged.

CPLB Example

```
CPLB R6,BLOCK2
```

Assume that register 6 contains X'0002'. CPLB places the Level Status Block for interrupt level 2 into the 22-byte storage area that begins at BLOCK2.

Copy Operand1 Key (CPOOK) (4955 Processor Only)

This instruction copies the operand 1 key field (bits 5–7) within the Address Key Register (AKR) to bits 13–15 of either a word in storage or a register.

Name	*Operation*	*Operand*
[label]	CPOOK	addr4 reg

If you code the *addr4* operand, CPOOK copies the contents of bits 5–7 from the AKR into bits 13–15 of the word defined by *addr4*.

If you code the *reg* operand, CPOOK copies the contents of bits 5–7 from the AKR into bits 13–15 of the specified register.

Indicators

All indicators are unchanged.

CPOOK Example

```
CPOOK R4
```

Assume that the AKR contains X'0120':

AKR

0000 0001	0010 0000

0 15

After execution of CPOOK, register 4 contains:

R4

0000 0000	0000 0001

0 15

The operand 1 key field within the address key register (AKR), has been copied into bits 13–15 while bits 0–12 have been set to zero. The AKR remains unchanged.

Copy Operand2 Key (CPOTK) (4955 Processor Only)

This instruction copies the operand 2 key field (bits 9–11) within the Address Key Register (AKR) into bits 13–15 of either a word in storage or a register.

Name	*Operation*	*Operand*
[label]	CPOTK	addr4 reg

If you code the *addr4* operand, CPOTK copies the contents of bits 9–11 from the AKR into bits 13–15 of the word defined by *addr4*.

If you code the *reg* operand, CPOTK copies the contents of bits 9–11 from the AKR into bits 13–15 of the specified register.

Indicators

All indicators are unchanged.

CPOTK Example

```
CPOTK R4
```

Assume that the AKR contains X'0120':

AKR

0000 0001	0010 0000

0 15

After execution of CPOTK register 4 contains:

R4

0000 0000	0000 0010

0 15

The operand 2 key field within the address key register (AKR) has been copied into bits 13–15 while bits 0–12 have been set to zero. The AKR remains unchanged.

Copy Processor Status and Reset (CPPSR)

This instruction places the contents of the Processor Status Word into a specified word in storage and resets bits 0–12 of the PSW.

Name	*Operation*	*Operand*
[label]	CPPSR	addr4

CPPSR stores the Processor Status Word in the address specified by *addr4*. (For the format of the Processor Status Word, see "Registers" in Chapter 3.)

CPPSR resets bits 0–12 of the PSW, leaving bits 13–15 unchanged.

Indicators

All indicators are unchanged.

CPPSR Example

```
CPPSR 12Ø(R2,64)*
```

In this example, here is how the address is calculated: The contents of register 2, plus 64, form an address. The contents of storage at that location, plus 120, form the address where the Processor Status Word is to be stored.

Copy Segmentation Register (CPSR) (4955 Processor Only)

This instruction places the contents of a specified segmentation register into a doubleword in storage.

Name	*Operation*	*Operand*
[label]	CPSR	reg, addr4

For *reg*, code the register that defines—in the following form—the segmentation register whose contents you want to store:

Bits 0–4 the 5 high-order bits of the logical storage address
Bits 5–7 the address key
Bits 8–15 zeros

For *addr4*, code the address of the doubleword into which the segmentation register is to be stored. CPSR copies the register into the doubleword in the following form:

Bits 0–12 physical segment address
Bit 13 if 1, the contents of the segmentation register are valid; if 0, any attempt to use this register results in program check.
Bit 14 if 1, the block is read-only; any attempt to write into the block while the processor is in problem state results in program check. Bit 14 is ignored when the processor is in supervisor state or during a cycle-steal access.
Bits 15–31 zeros

CPSR leaves the segmentation register unchanged.

Indicators

All indicators are unchanged.

CPSR Example

```
CPSR R3,DWORD
```

Assume that register 3 contains:
0110 1100 0000 0000
and segmentation register 108 contains:
1110 0001 1101 0000
CPSR places the following result into the doubleword at DWORD:
1110 0001 1101 0000 0000 0000 0000 0000

Copy Storage Key (CPSK) (4955 Processor Only)

This instruction places the contents of a specified storage key into a byte in storage.

Name	*Operation*	*Operand*
[label]	CPSK	reg, addr4

For protection purposes, storage is divided into blocks of 2048 bytes. Each block has associated with it a *storage key register*.

The *reg* operand defines the general purpose register that contains, in the following form, the number of the storage key register to be copied:

Bits 0–4 the block number in main storage (0–31)
Bits 5–15 zeros

CPSK places the storage key into the byte at *addr4*, in the following form:

Bits 0–3 zeros
Bits 4–6 the value of the storage key
Bit 7 read-only bit

The storage key remains unchanged.

Indicators

All indicators are unchanged.

CPSK Example

```
CPSK R4,KEY
```

Assume that R4 contains:
1010 1000 0000 0000
Because bits 0–4 of R4 contain X'15', CPSK copies the storage key register for block 21. Assume that the value of this key is 4 and the read-only bit is on. CPSK places X'09' into the byte at KEY.

The storage key register and register 4 remain unchanged.

Diagnose (DIAG)

This machine-dependent instruction controls and tests various hardware functions. It is not intended for use in problem programs or supervisor programs.

Name	*Operation*	*Operand*
[label]	DIAG	ubyte

For the specific meaning of the *ubyte* field, and for a discussion of the instruction's diagnostic functions, refer to the processor description manual for your processor. See the Preface of this manual for titles and order numbers.

Disable (DIS)

This instruction disables:

- Storage protection,
- Equate operand spaces,
- Address translator, or
- Summary mask,

depending on the value you specify for a 1-byte mask.

Name	*Operation*	*Operand*
[label]	DIS	ubyte

The mask has the following format:

Bits 0–3	unused, must be coded as zeros
Bit 4	storage protection
Bit 5	equate operand spaces
Bit 6	address translator—this bit is ignored if the translator is not fitted on your system.
Bit 7	summary mask

Notes.

1. On the 4953 Processor, bits 0–6 of the mask are not used.
2. This instruction is not interruptable.

Indicators

All indicators are unchanged.

DIS Example

```
DIS X'Ø9'
```

In this example, bits 4 and 7 of the DIS mask are ON; storage protection and the summary mask are disabled.

Enable (EN)

The Enable (EN) instruction enables:

- Storage protection,
- Equate operand spaces,
- Address translator, or
- Summary mask,

depending on the value you specify for a 1-byte mask.

Name	*Operation*	*Operand*
[label]	EN	ubyte

The mask has the following format:

Bits 0–3 unused, must be coded as zeros
Bit 4 storage protection
Bit 5 equate operand spaces
Bit 6 address translator—this bit is ignored if the translator is not fitted on your system.
Bit 7 summary mask

Note. On the 4953 Processor, bits 0–6 of the mask are not used.

Indicators

All indicators are unchanged.

EN Example

```
EN X'Ø4'
```

In this example, bit 5 of the EN mask is ON; equate operand spaces is enabled.

Interchange Operand Keys (IOPK) (4955 Processor Only)

This instruction interchanges the contents of operand 1 key and operand 2 key in the Address Key Register.

Name	*Operation*	*Operand*
[label]	IOPK	

Note. For information about the contents of the Address Key Register see *IBM 4955 Processor and Processor Features Description*, GA34-0021.

Indicators

All indicators are unchanged.

IOPK Example

```
IOPK
```

Assume that the AKR contains X'0120'. After execution of IOPK, the AKR would contain X'0210'.

Level Exit (LEX)

This instruction causes the processor to exit the current level.

Name	*Operation*	*Operand*
[label]	LEX	[ubyte]

For the *ubyte* operand, you can optionally code a 1-byte unsigned absolute value or expression. The processor ignores this value, so you can use it for a flag or indicator.

Note. *ubyte* defaults to zero when it's not coded.

LEX does one of two things:

- If no interrupts are pending, it places the processor into wait state.
- If there are interrupts waiting, the one with the highest priority is given control.

Indicators

All indicators are unchanged.

LEX Example

```
LEX 175
```

Assume that the processor encountered this LEX while it was running on level 0, and there are interrupts waiting on levels 1 and 3. The processor exits level 0, and begins servicing the first interrupt for level 1.

Operate I/O (IO)

This instruction initiates input/output operations from the processor.

Name	*Operation*	*Operand*
[label]	IO	longaddr

For the *longaddr* operand, code the address of the Immediate Device Control Block (IDCB) that defines the I/O operation you want to perform.
For a detailed discussion of the IDCB, refer to the processor description manual for your processor. See the Preface of this manual for titles and order numbers.

Indicators

This instruction sets a condition code using the even, carry, and overflow indicators. See "Other Uses of Indicators" in Chapter 3 of this manual.

Set Address Key Register (SEAKR) (4955 Processor Only)

This instruction sets a specified value in the Address Key Register (AKR) (bits 0–15) from a storage location or a register.

Name	*Operation*	*Operand*
[label]	SEAKR	addr4 reg

If you code the *addr4* operand the SEAKR instruction loads the contents of this word into bits 0–15 of the AKR.

If you code the *reg* operand the contents of the specified register are placed in bits 0–15 of the AKR.

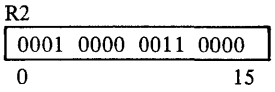
Indicators

All indicators are unchanged.

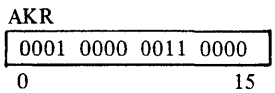
SEAKR Example

```
SEAKR R2
```

Assume that R2 contains X'1030':



After execution of SEAKR the AKR contains:



Register 2 remains unchanged.

Set Console Data Lights (SECON)

This instruction places the contents of a specified register into the console data lights.

Name	*Operation*	*Operand*
[label]	SECON	reg

Note. If your processor does not have the full-function console, SECON is treated as a no-operation.

Indicators

All indicators are unchanged.

Set Instruction Space Key (SEISK) (4955 Processor Only)

This instruction sets the instruction space key field (bits 13–15) within the Address Key Register (AKR) from the contents of the word (bits 13–15) defined by the *addr4* or *reg* operand. Bits 0–12 in the AKR remain unchanged.

Name	*Operation*	*Operand*
[label]	SEISK	addr4 reg

If you code the *addr4* operand the SEISK instruction loads bits 13–15 of this word into bits 13–15 of the AKR.

If you code the *reg* operand bits 13–15 of the specified register are placed in bits 13–15 of the AKR.

Indicators

All indicators are unchanged.

SEISK Example

```
SEISK R4
```

Assume that R4 contains X'0002':

R4

0000	0000	0000	0010
0			15

After execution of SEISK bits 13–15 of the AKR contain:

AKR

xxxx	xxxx	xxxx	x010
0			15

Bits 0–12 of the AKR remain unchanged. Register 4 also remains unchanged.

Set Interrupt Mask Register (SEIMR)

This instruction loads the Interrupt Level Mask Register from a word in storage.

Name	*Operation*	*Operand*
[label]	SEIMR	addr4

For *addr4*, code the address of the location that contains the value to be loaded into the Interrupt Level Mask Register.

Each bit in the register corresponds to an interrupt level—bit 0 corresponds to level 0, bit 1 corresponds to level 1, and so on. If the bit for a given level is 1, that level is enabled and can accept interrupts.

SEIMR leaves the word in storage unchanged.

Note. Set to zero any bits that correspond to nonexistent interrupt levels.

Indicators

All indicators are unchanged.

SEIMR Example

```
SEIMR (R2)
```

In this example, assume that the word whose address is in R2 contains:
1011 0000 0000 0000
SEIMR loads this value into the Mask Register, leaving the word in storage unchanged. Now levels 0, 2, and 3 can accept interrupts. All other levels are disabled.

Set Level Status Block (SELB)

This instruction loads a specified Level Status Block from a word-aligned 22-byte storage area that you define.

Name	*Operation*	*Operand*
[label]	SELB	reg, addr4

The *reg* operand defines the interrupt level whose Level Status Block is to be loaded. Load *reg* so that the level is specified in bits 12 through 15, with bits 1–11 containing zeros. Bit 0 is the inhibit-trace-interrupt bit. SELB ignores bits that correspond to nonexistent levels.

The *addr4* operand defines the first word of a 22-byte (11-word) storage area that is to be loaded into the Level Status Block. The storage area has the following format:

Main storage address (LSB pointer)

Instruction address register
Address key register *
Level status register
Register 0
Register 1
Register 2
Register 3
Register 4
Register 5
Register 6
Register 7

0 15

*4955 processor *only*

The 22-byte storage area remains unchanged.

Note. If this instruction turns off bit 8 of the Level Status Register, the processor leaves supervisor state. This instruction is the only way to exit supervisor state.

Indicators

The indicators reflect the contents of the LSR that this instruction loaded into word 2 of the Level Status Block.

SELB Example

```
SELB R4,BLOCK3
```

Assume that R4 contains X'0003'. SELB places the contents of the 22-byte storage area that begins at BLOCK3 *into* the Level Status Block for level 3. If the LSR trace bit is on in the LSB, trace interrupts can occur after SELB has executed.

Set Operand1 Key (SEOOK) (4955 Processor Only)

This instruction sets the operand 1 key field (bits 5–7) within the Address Key Register (AKR) from the contents of the word (bits 13–15) defined by the *addr4* or *reg* operand. Bits 0–4 and 8–15 of the AKR remain unchanged.

Name	*Operation*	*Operand*
[label]	SEOOK	addr4 reg

If you code the *addr4* operand the SEOOK instruction loads bits 13–15 of this word into bits 5–7 of the AKR.

If you code the *reg* operand bits 13–15 of the specified register are placed in bits 5–7 of the AKR.

Indicators

All indicators are unchanged.

SEOOK Example

```
SEOOK R3
```

Assume R3 contains X'0002':

R3

0000 0000 0000 0010
0 15

After execution of SEOOK bits 5–7 of the AKR contain:

AKR

xxxx x010 xxxx xxxx
0 15

Bits 0–4 and 8–15 of the AKR remain unchanged. Register 3 also remains unchanged.

Set Operand2 Key (SEOTK) (4955 Processor Only)

This instruction sets the operand 2 key field (bits 9–11) within the Address Key Register (AKR) from the contents of the word (bits 13–15) defined by the *addr4* or *reg* operand. Bits 0–8 and 12–15 of the AKR remain unchanged.

Name	*Operation*	*Operand*
[label]	SEOTK	addr4 reg

If you code the *addr4* operand the SEOTK instruction loads bits 13–15 of this word into bits 9–11 of the AKR.

If you code the *reg* operand, bits 13–15 of the specified register are placed in bits 9–11 of the AKR.

Indicators

All indicators are unchanged.

SEOTK Example

```
SEOTK R4
```

Assume register 4 contains X'0001':

R4

0000 0000 0000 0001
0 15

After execution of SEOTK bits 9–11 of the AKR contain:

AKR

xxxx xxxx x001 xxxx
0 15

Bits 0–8 and 12–15 remain unchanged. Register 4 also remains unchanged.

Set Segmentation Register (SESR) (4955 Processor Only)

This instruction places the contents of a doubleword in storage into a specified segmentation register.

Name	*Operation*	*Operand*
[label]	SESR	reg, addr4

For *reg*, code the register that defines the segmentation register you want to load. See CPSR for the form of this register.

For *addr4*, code the address of the doubleword whose contents are to be loaded into the segmentation register. See CPSR for the form of this doubleword.

SESR loads the doubleword into the segmentation register, leaving the doubleword unchanged.

Indicators

All indicators are unchanged.

SESR Example

```
SESR R5,(R3)
```

Assume that register 5 contains:
1100 0011 0000 0000
and doubleword whose address is in register 3 contains:
0010 1111 0010 0000 0000 0000 0000 0000
SESR places the following result into segmentation register 195:
0010 1111 0010 0000 0000 0000 0000 0000

Set Storage Key (SESK) (4955 Processor Only)

This instruction places a specified value into a storage key.

Name	*Operation*	*Operand*
[label]	SESK	reg, addr4

The *reg* operand defines the general purpose register that contains, in the following form, the number of the storage key register to be loaded:

Bits 0–4 the block number in main storage (0–31)
Bits 5–15 zeros

The value that SESK loads into the storage key is in the following form:

Bits 0–3 zeros
Bits 4–6 the value of the storage key
Bit 7 read-only bit

The contents of *reg* and *addr4* remain unchanged.

Indicators

All indicators are unchanged.

SESK Example

```
SESK R7,(R1)
```

Assume that register 7 contains:
1101 1000 0000 0000

Because bits 0–4 of R7 contain decimal 27, SESK loads the storage key register 27. Assume that the byte whose address is in register 1 contains X'07'. SESK loads this 8-bit value into storage key 27. The contents of R7, R1, and the byte in storage remain unchanged.

Floating-Point Instructions

Note. To use the floating-point instructions you must have either the floating-point hardware feature or the floating-point emulator.

Floating-Point Number Representation

Each floating-point number is represented as a combination of two items: a numeric fraction and a power of 16 by which the fraction is multiplied. The power of 16 is called the *characteristic*, by analogy with logarithms. For instance, the number 1 is represented by $1/16 \times 16^1$. The $1/16$ is stored as a hexadecimal fraction: 0.1_{16} The characteristic is written in *excess 64* notation, which means that every characteristic is 64_{10} greater than the power actually represented. Thus, the characteristic +1 is represented as:

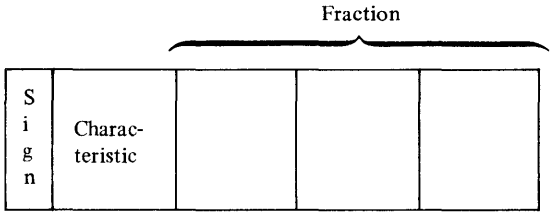
$=1+64 = 65_{10} = 41_{16}$

The characteristic −1 becomes:

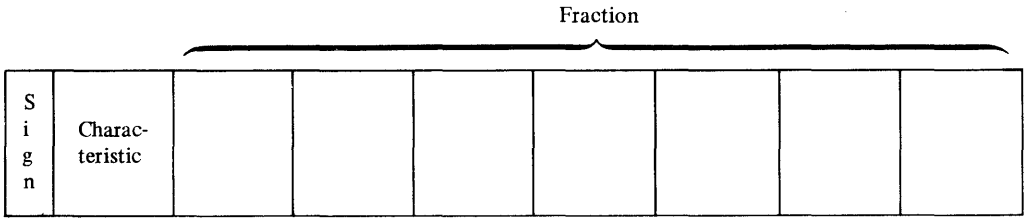
$-1+64 = 63_{10} = 3F_{16}$

The excess 64 method is used to avoid the need of a sign for the characteristic.

The processor permits two types of floating-point numbers, called *single-precision* and *double-precision.* In each, the excess 64 exponent is contained in bits 1–7 of the first byte (bit 0 is the sign of the number). In a single-precision number, the fractional part consists of six hexadecimal digits contained in the next three bytes.

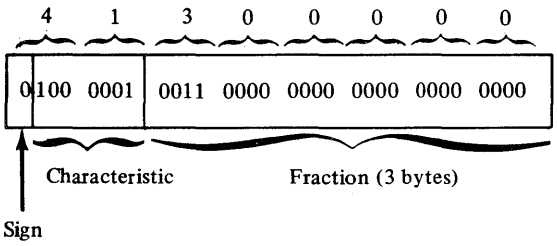


In a double-precision number, the fractional part consists of 14 hexadecimal digits contained in the next seven bytes.

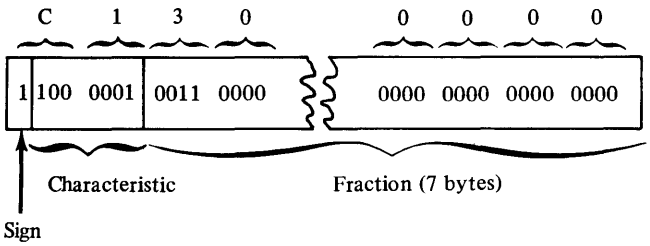


A single-precision number, therefore, occupies two words, and a double-precision number four words.

The number +3 would be represented as 4130 0000 in (single-precision) floating point, and would be stored as:



The number −3 would be represented as C130 0000 0000 0000 in (double-precision) floating point, and would be stored as:



Here are examples of some single-precision floating-point representations:

Decimal	*Floating-Point*
0	0000 0000
1	4110 0000
9	4190 0000
16	4210 0000
4096	4410 0000
−1	C110 0000
−15	C1F0 0000
0.5	4080 0000
0.001	3E41 8937

And here are some double-precision numbers:

Decimal	*Floating-Point*
0	0000 0000 0000 0000
2	4120 0000 0000 0000
12345678912345	4BB3 A73C E5B5 9000
0.1	4019 9999 9999 999A
−15	C1F0 0000 0000 0000

When a floating-point number has no leading hexadecimal zeros in its fraction, it is said to be *normalized.* Note that a normalized floating-point number might have as many as three leading binary zeros in its fraction. For example, the floating-point number 4120 0000 is normalized, because the first hexadecimal digit of the fraction (200000) is not zero. The binary representation of the first digit of the fraction, however, is 0010, which has two leading binary zeros. Define Constant (DC) entries in assembler language are always converted to normalized form, and floating-point numbers in storage are assumed to be normalized.

Floating-point representation can express decimal values ranging from about 5.4×10^{-79} to about 7.2×10^{75}.

Floating-point Registers and Instructions

Each interrupt level has four 64-bit floating-point registers, numbered 0, 1, 2, and 3. All floating-point arithmetic and compare instructions use at least one of these registers, placing results (from arithmetic instructions) in the register defined by the second operand.

The entire set of floating-point instructions is available for both single-precision and double-precision operands. When you code a single-precision arithmetic instruction—FA, FD, or FS—all operands and results are 32-bit floating-point values. The rightmost 32 bits of the floating-point registers do not

participate and are unchanged. The product in Floating Multiply (FM) is 64 bits, and occupies a full register. When you code a single-precision move instruction—FMV or FMVC—the rightmost 32 bits of the floating-point register are set to zero. When you code a double-precision instruction—such as FAD, FDD, or FSD—all operands and results occupy 64 bits.

Copy Floating Level Block (CPFLB)

This instruction places the contents of the floating-point registers for a specified level into a word-aligned 32-byte storage area. *This is a privileged instruction.*

Name	*Operation*	*Operand*
[label]	CPFLB	reg, addr4

For *reg*, code the general-purpose register that contains, in bits 12–15, the interrupt level whose floating-point registers you want to store. Bits 0–11 of *reg* must contain zeros.

For *addr4*, code the address of the first byte of the 32-byte storage area where the registers are to be stored.

The floating-point registers and *reg* are unchanged.

Indicators

All indicators are unchanged.

CPFLB Example

```
CPFLB R6,FREGS
```

Assume that R6 contains X'0001'. CPFLB copies the contents of the floating-point registers for level 1 into the 32-byte storage area that begins at FREGS.

Floating Add (FA)

This instruction adds two single-precision floating-point values, and places the normalized result into a floating-point register.

Name	*Operation*	*Operand*
[label]	FA	addr4, freg freg, freg

The floating-point value specified by the first operand is added to the contents of the *freg* specified by the second operand. The first operand is unchanged.

Indicators

The even, carry, and overflow indicators are reset. The overflow indicator is turned on if an underflow or overflow occurs. If there is an underflow, the even indicator is also turned on. The carry indicator is reset, and the remaining indicators reflect the result.

FA Example

```
FA FR3,FRØ
```

Assume that:

- Bits 0–31 of FR3 contain 4150 0000 (the floating-point hexadecimal representation of decimal 5).
- Bits 0–31 of FR0 contain 41C0 0000 (the floating-point hexadecimal representation of decimal 12).

FA adds the two values, and places 4211 0000 (the floating-point hexadecimal representation of decimal 17) into FR0. FR3 is unchanged, and bits 32–63 of FR0 contain zeros.

Floating Add Double (FAD)

This instruction adds two double-precision floating-point numbers, and places the normalized result into a floating-point register.

Name	*Operation*	*Operand*
[label]	FAD	addr4, freg freg, freg

The floating-point value specified by the first operand is added to the contents of the *freg* specified by the second operand. The first operand is unchanged.

Indicators

The even, carry, and overflow indicators are reset. The overflow indicator is turned on if an underflow or overflow occurs. If there is an underflow, the even indicator is also turned on. The carry indicator is reset, and the remaining indicators reflect the result.

FAD Example

```
FAD FLOAT,FR2
```

Assume that:

- The 64 bits at FLOAT contain 41C0 0000 0000 0000 (the double-precision, floating-point, hexadecimal representation of decimal 12).
- The 64 bits in FR2 contain 422F 0000 0000 0000 (the double-precision, floating-point, hexadecimal representation of decimal 47).

FAD adds the two values, and places 423B 0000 0000 0000 (the double-precision, floating-point, hexadecimal representation of decimal 59) into FR2. FLOAT is unchanged.

Floating Compare (FC)

This instruction compares two single-precision numbers, and sets indicators to reflect the result.

Name	*Operation*	*Operand*
[label]	FC	freg, freg

FC compares bits 0–31 of the two registers. See "Compare Instructions" for a discussion of the compare process. FC leaves both operands unchanged.

Indicators

The even, carry, and overflow indicators are reset. If an underflow or overflow occurs, the overflow indicator is turned on. If there is an underflow, the even indicator is also turned on. The remaining indicators reflect the result.

FC Example

```
FC FRØ,FR1
```

Assume that:

- Bits 0–31 of FR0 contain 41F0 0000, the floating-point equivalent of decimal 15.
- Bits 0–31 of FR1 contain C1F0 0000, the floating-point equivalent of decimal −15.

FC compares the two values, and sets an *arithmetically less than* condition.

Floating Compare Double (FCD)

This instruction compares two double-precision floating-point numbers, and sets indicators to reflect the result.

Name	*Operation*	*Operand*
[label]	FCD	freg, freg

FCD compares the two registers. See "Compare Instructions" for a discussion of the compare process. FCD leaves both operands unchanged.

Indicators

The even, carry, and overflow indicators are reset. If an underflow or overflow occurs, the overflow indicator is turned on. If there is an underflow, the even indicator is also turned on. The remaining indicators reflect the result.

FCD Example

```
FCD FR3,FR2
```

Assume that:

- FR3 contains 4211 0000 0000 0000, the floating-point equivalent of decimal 17.
- FR2 contains 4111 9999 9999 999A, the floating-point equivalent of decimal 1.1.
- FCD compares the two values, and sets an *arithmetically less than* condition.

Floating Diagnose (FDIAG)

This instruction tests various functions of the floating-point hardware. It is not intended for use in problem programs or supervisor programs.

Name	*Operation*	*Operand*
[label]	FDIAG	

For a discussion of the instruction's diagnostic functions, see *IBM 4955 Processor and Processor Features Description,* GA34-0021.

Floating Divide (FD)

This instruction divides one single-precision floating-point number into another.

Name	*Operation*	*Operand*
[label]	FD	addr4, freg freg, freg

Bits 0–31 of the *freg* defined by the second operand are divided by the 32-bit value defined by the first operand. FD places the result into bits 0–31 of the second operand, leaving the first operand unchanged. Bits 32–63 of the second operand are unchanged. FD saves no remainder.

Note. If you try to divide by zero, neither operand is altered.

Indicators

The even, carry, and overflow indicators are reset, then:

If this occurs:	*These indicators are turned on:*
Overflow	Overflow
Underflow	Overflow & Even
Attempt to divide by zero	Carry & Overflow

The remaining indicators are set to reflect the result.

FD Example

```
FD FLOAT,FR2
```

Assume that:

- Bits 0–31 of FR2 contain 4310 2000, the floating-point equivalent of decimal 258.
- The 32 bits at FLOAT contain 4256 0000, the floating-point equivalent of decimal 86.

FD divides: 4310 2000 divided by 4256 0000 = 4130 0000 (in decimal it would be 258 divided by 86 = 3), and places the result into bits 0–31 of FR2. Bits 32–63 of FR2 are unchanged.

Floating Divide Double (FDD)

This instruction divides one double-precision floating-point number into another.

Name	*Operation*	*Operand*
[label]	FDD	addr4, freg freg, freg

The contents of the *freg* defined by the second operand are divided by the 64-bit value defined by the first operand. FDD places the result into the second operand, leaving the first operand unchanged. FDD saves no remainder.

Note. If you try to divide by zero, neither operand is altered.

Indicators

See "Indicators" under FD.

FDD Example

```
FDD FRØ,FR3
```

Assume that:

- FR0 contains 4421 C500 0000 0000, the floating-point equivalent of decimal 8645.
- FR3 contains 472B 2D30 9000 0000, the floating-point equivalent of decimal 45,273,865.

FDD divides:
472B 2D30 9000 0000 divided by 4421 C500 0000 0000 =
4414 7500 0000 0000
(in decimal it would be 45,273,865 divided by 8645 = 5237), and places the result into FR3.

Floating Move (FMV)

This instruction moves a single-precision floating-point number:

- From storage to bits 0–31 of a floating-point register (setting bits 32–63 to zero),
- From bits 0–31 of one floating-point register to bits 0–31 of another (setting bits 32–63 of the destination to zero), or
- From bits 0–31 of a floating-point register to a 4-byte storage location.

Name	*Operation*	*Operand*
[label]	FMV	addr4, freg freg, freg freg, addr4

Indicators

The even, carry, and overflow indicators are reset. The remaining indicators reflect the new contents of the second operand.

FMV Example

```
FMV FLOAT,FR1
```

Assume that:

- FLOAT contains 4501 3C76.
- Bits 0–31 of FR1 contain 3F3B 249D.

FMV moves the contents of FLOAT to bits 0–31 of FR1. After FMV, both operands contain 4501 3C76. Bits 32–63 of FR1 contain zeros.

Floating Move Double (FMVD)

This instruction moves a double-precision floating-point number:

- From storage to a floating-point register,
- From one floating-point register to another, or
- From a floating-point register to an 8-byte storage location.

Name	*Operation*	*Operand*
[label]	FMVD	addr4, freg freg, freg freg, addr4

Indicators

The even, carry, and overflow indicators are reset. The remaining indicators reflect the new contents of the second operand.

FMVD Example

Note. *ubyte* defaults to zero when it's not coded.

```
FMVD FR3,FLOAT
```

Assume that:

- FR3 contains 4112 5CE2 3010 0000.
- FLOAT contains 4200 3458 CDF1 2000.

FMVD moves the contents of FR3 to FLOAT. After FMVD, both operands contain 4112 5CE2 3010 0000.

Floating Move and Convert (FMVC)

This instruction does one of the following:

- Converts an integer to a single-precision floating-point number, loading it into bits 0–31 of a floating-point register.
- Converts a single-precision floating-point number to an integer, placing it into a 1-word storage location.

Name	*Operation*	*Operand*
[label]	FMVC	addr4, freg freg, addr4

If you code the *addr4,freg* form, FMVC converts the signed, 2-byte integer at *addr4* to a 32-bit floating-point number, then places it into bits 0—31 of *freg*. Bits 32–63 of *freg* are set to zero.

If you code the *freg,addr4* form, FMVC converts bits 0–31 of the floating-point number in *freg* to a signed 2-byte integer, then stores it at *addr4*. Any fractional portion of the floating-point number is truncated.

The first operand is unchanged.

Indicators

The even, carry, and overflow indicators are reset. If you coded *addr4,freg*, the remaining indicators are set to reflect the new contents of the second operand. If you specified the *freg,addr4* form, the indicators are set as follows: If the converted number is larger than $+2^{15}-1$ or less than -2^{15}, the carry indicator is turned on. In this case, the value stored is either the largest ($+2^{15}-1$) or the smallest (-2^{15}) representable number. The remaining indicators reflect the new contents of the second operand.

FMVC Examples

```
FMVC (R5),FR2
```

Assume that the word whose address is in R5 contains X'0018' (the equivalent of decimal 24). FMVC converts this value to floating-point 4218 0000, and places it into bits 0–31 of FR2. Bits 32–63 of FR2 contain zeros.

```
FMVC FRØ,INT+2
```

Assume that FR0 contains 41C0 0000. FMVC converts this value to X'000C' (the equivalent of decimal 12), and places it into the word that is 2 bytes past INT.

Floating Move and Convert Double (FMVCD)

This instruction does one of the following:

- Converts an integer to a double-precision floating-point number, loading it into a floating-point register.
- Converts a double-precision floating-point number to an integer, placing it into a doubleword storage location.

Name	*Operation*	*Operand*
[label]	FMVCD	addr4, freg freg, addr4

If you code the *addr4,freg* form, FMVCD converts the signed, 4-byte integer at *addr4* to a 64-bit floating-point number, then places it into *freg*.

If you code the *freg,addr4* form, FMVCD converts the 64-bit floating-point number in *freg* to a signed 4-byte integer, then stores it at *addr4*. Any fractional portion of the floating-point number is truncated.

The first operand is unchanged.

Indicators

The even, carry, and overflow indicators are reset. If you coded *addr4,freg*, the remaining indicators are set to reflect the new contents of the second operand. If you specified the *freg,addr4* form, the indicators are set as follows: If the converted number is larger than $+2^{31}-1$ or less than -2^{31}, the carry indicator is turned on. In this case the value stored is either the largest ($+2^{31}-1$) or the smallest (-2^{31}) representable number. The remaining indicators reflect the new contents of the second operand.

FMVCD Examples

```
FMVCD (R3)*,FR3
```

R3 contains the address of a storage location. Assume that the doubleword whose address is in that location contains X'FFFF EFFF' (the equivalent of decimal −4097). FMVCD converts this value to floating-point C410 0100 0000 0000, placing it in FR3.

```
FMVCD FRØ,(R6)
```

Assume that FR0 contains 42FF 0000 0000 0000. FMVCD converts this value to X'00FF' (the equivalent of decimal 255), placing it into the storage location whose address is in R6.

Floating Multiply (FM)

This instruction multiplies one single-precision floating-point number by another.

Name	*Operation*	*Operand*
[label]	FM	addr4, freg freg, freg

The 32-bit value defined by the first operand is multiplied by bits 0–31 of the *freg* defined by the second operand. FM places the result into bits 0–63 of the second operand, leaving the first operand unchanged.

Indicators

The even, carry, and overflow indicators are reset. The overflow indicator is turned on if an overflow or underflow occurs. If there is an underflow, the even indicator is also turned on. The remaining indicators are set to reflect the result.

FM Example

```
FM FR1,FR3
```

Assume that:

- Bits 0–31 of FR1 contain 4120 0000, the floating-point equivalent of decimal 2.
- Bits 0–31 of FR3 contain 4160 0000, the floating-point equivalent of decimal 6.

FM multiplies the two values, placing the result, 41C0 0000 0000 0000 (the floating-point equivalent of decimal 12), in all 64 bits of FR3. FR1 is unchanged.

Floating Multiply Double (FMD)

This instruction multiplies one double-precision floating-point number by another.

Name	*Operation*	*Operand*
[label]	FMD	addr4, freg freg, freg

The 64-bit value defined by the first operand is multiplied by the *freg* defined by the second operand. FMD places the result into the second operand, leaving the first operand unchanged.

Indicators

The even, carry, and overflow indicators are reset. The overflow indicator is turned on if an overflow or underflow occurs. If there is an underflow, the even indicator is turned on. The remaining indicators are set to reflect the result.

FMD Example

```
FMD FLOAT+4,FR2
```

Assume that:

- The 64 bits at FLOAT+4 contain 4219 0000 0000 0000, the floating-point equivalent of decimal 25.
- FR2 contains 4140 0000 0000 0000, the floating-point equivalent of decimal 4.

FM multiplies the two values, placing the result, 4264 0000 0000 0000 (the floating-point equivalent of decimal 100), into FR2. The 64 bits at FLOAT+4 are unchanged.

Floating Subtract (FS)

This instruction subtracts one single-precision floating-point number from another.

Name	*Operation*	*Operand*
[label]	FS	addr4, freg freg, freg

The 32-bit value specified by the first operand is subtracted from bits 0–31 of the *freg* defined by the second operand. FS places the normalized result into bits 0–31 of the second operand, leaving the first operand unchanged. Bits 32–63 of the second operand are set to zero.

Indicators

The even, carry, and overflow indicators are reset. If an underflow or overflow occurs, the overflow indicator is turned on. If there is an underflow, the even indicator is also turned on. The remaining indicators reflect the result.

FS Example

```
FS FR1,FR2
```

Assume that:

- Bits 0–31 of FR1 contain 4150 0000, the floating-point equivalent of decimal 5.
- Bits 0–31 of FR2 contain 4211 0000, the floating-point equivalent of decimal 17.

FS subtracts FR1 from FR2, leaving 41C0 0000 (the floating-point equivalent of decimal 12) in FR2. Bits 32–63 of FR2 are unchanged, as is FR1.

Floating Subtract Double (FSD)

This instruction subtracts one double-precision floating-point number from another.

Name	*Operation*	*Operand*
[label]	FSD	addr4, freg freg, freg

The 64-bit value specified by the first operand is subtracted from the *freg* defined by the second operand. FSD places the normalized result into the second operand, leaving the first operand unchanged.

Indicators

The even, carry, and overflow indicators are reset. If an underflow or overflow occurs, the overflow indicator is turned on. If there is an underflow, the even indicator is also turned on. The remaining indicators reflect the result.

FSD Example

```
FSD FR1,FR3
```

Assume that:

- FR1 contains 4210 0000 0000 0000, the floating-point equivalent of decimal 16.
- FR3 contains 4310 0000 0000 0000, the floating-point equivalent of decimal 256.

FSD subtracts FR1 from FR3, leaving 42F0 0000 0000 0000 (the floating-point equivalent of decimal 240) in FR3. FR1 is unchanged.

Set Floating Level Block (SEFLB)

This instruction loads the floating-point registers for a specified level from a word-aligned 32-byte storage area. *This is a privileged instruction.*

Name	*Operation*	*Operand*
[label]	SEFLB	reg, addr4

For *reg*, code the general-purpose register that contains, in bits 12–15, the interrupt level whose floating-point registers you want to load. Bits 0–11 of *reg* must contain zeros.

For *addr4*, code the address of the first byte of the 32-byte storage area that the registers are to be loaded from. The contents of the storage location and *reg* are unchanged.

Indicators

All indicators are unchanged.

SEFLB Example

```
SEFLB R0,(R1)
```

Assume that R0 contains X'0003'. SEFLB loads the floating-point registers for level 3 from the 32-byte storage area whose starting address is in register 1.

Chapter 5. Assembler Instructions

Section Contents

Establishing Symbolic Representation

Symbols greatly reduce programming effort and errors. You can define symbols to represent storage addresses, displacements, constants, registers, and other elements that make up the assembler language.

Some symbols represent absolute values, while others represent relocatable address values. Relocatable addresses are associated with:

- Instructions
- Constants
- Storage areas

You can use these defined symbols in the operand fields of instruction statements to refer to the instruction, constant, or area represented by the symbol.

Assigning Values to Symbols

The EQU and EQUR instructions assign values to symbols:

- EQU—for symbols other than registers
- EQUR—for symbols that represent registers

EQU—Equate Symbol

EQU assigns absolute or relocatable values to symbols. You can use it for the following purposes:

- To assign single absolute values to symbols.
- To assign the values of previously defined symbols or expressions to new symbols, thus allowing you to use different symbolic names for different purposes.
- To compute expressions whose values are unknown at coding time or difficult to calculate. The value of the expression is then assigned to a symbol.

You can code the EQU instruction anywhere in a source module after any source macro definitions you have specified. Note, however, that the EQU instruction initiates an unnamed control section (private code) if you code it before the first control section (initiated by a START or CSECT instruction).

The format of the EQU instruction statement is:

Name	*Operation*	*Operand*
label	EQU	expression

The label field can contain any ordinary symbol. *Expression* represents a value. It must always be specified and can have a relocatable or absolute value in the range –65536 through +65535. The assembler evaluates the expression internally as a signed 32-bit number. Only the rightmost 16 bits are retained.

You must define all symbols appearing in the expressions in previously coded instructions—instructions that *physically precede* this EQU in the source module.

The assembler assigns an absolute or relocatable value to the symbol in the name field (the label) of the EQU instruction. The length of the symbol is the length attribute of the leftmost or only term in the operand.

The following examples indicate valid EQU statements and the value (absolute or relocatable) assigned to the symbol in the label field of each.

```
SECTA    START Ø
         :
FULL     DC    F'33'
AREA     DS    XL2ØØ
TO       DS    CL24Ø
FROM     DS    CL8Ø
         :
ADCONS   DC    A(X,Y,Z)
         :
A        EQU   X'FF'          ABSOLUTE
B        EQU   *+4            RELOCATABLE
C        EQU   A*1Ø           ABSOLUTE
D        EQU   FULL           RELOCATABLE
E        EQU   AREA+1ØØ       RELOCATABLE
F        EQU   TO             RELOCATABLE
G        EQU   FROM-TO        ABSOLUTE
H        EQU   ADCONS         RELOCATABLE
I        EQU   SECTA          RELOCATABLE
         :
         END
```

EQUR—Equate Register

EQUR defines a register symbol (that may be used in addition to the predefined register names) by assigning to the symbol the value of an absolute expression. You can code the EQUR instruction anywhere in a source module after the start of the program control section and after any other statement that defines symbols used in the absolute expression on the EQUR instruction. The EQUR instruction must precede all assembler and machine instructions that use the register symbol.

The format of the EQUR instruction is:

Name	*Operation*	*Operand*
label	EQUR	absolute expression

where *label* is an ordinary symbol given the value of the absolute expression (value must be in the range 0–7). Any symbols in the absolute expression must be previously defined (defined in statements coded prior to the EQUR instruction). The symbol is absolute and its length attribute is 1.

Note. All register specifications in machine instructions must contain a register symbol, which is either one of the predefined register symbols or has been defined in a preceding EQUR instruction. These register symbols may only appear in machine or assembler instructions as a register specification.

The following examples indicate valid EQUR instructions.

```
REG1     EQUR  1
REG2     EQUR  2
REG3     EQUR  3
REG4     EQUR A+B    (ASSUMING A=3 AND B=1, REG4=4.)
```

Defining Data

This section describes DC and DS instructions, used to define data constants and reserve main storage. You can code a label for these instructions and then refer to the data constant or storage area symbolically in the operands of machine and assembler instructions. The symbol used as a label represents the address of the constant or storage area—do not confuse it with the assembled object code for the constant or contents of the storage area. This data is generated, and storage is reserved at assembly time and used by the machine instructions at execution time.

DC—Define Constant

DC defines data constants needed for program execution. The DC instruction causes the assembler to generate (at assembly time) the binary representation of the data constant you specify, storing that value in a particular location in the assembled object module. One DC statement can generate a maximum of 65535 bytes of data (the product of the length and duplication factor must be less than or equal to 65535).

The DC instruction can generate the following types of constants:

- Binary constants, which define bit patterns
- Character constants (EBCDIC, ASCII, or PTTC/EBCD), which define character strings or messages
- Hexadecimal constants, which define hexadecimal numeric values
- Fixed-point constants, which define fixed-point numeric values
- Floating-point constants, which define floating-point numeric values
- Address constants, which define addresses or values resulting from expression evaluation
- Name constants, which define resource references

The format of the DC instruction is:

Name	*Operation*	*Operand*
[label]	DC	[dup] type[mods] { 'value' / (value) } [,opnd 2]...

The symbol in the name field represents the address of the left most byte of the assembled constant. The operand in a DC instruction consists of 4 subfields. The first 3 subfields describe the constant, and the fourth subfield specifies the nominal value of the constant to be generated.

Subfield	*Contents*
1	Duplication factor (optional)
2	Constant type (required)
3	Modifiers (optional)
4	Nominal value or values (required)

Rules for the DC Operand.

- The type subfield and the nominal value must always be specified.
- The duplication factor and modifier subfields are optional.
- When multiple operands are specified, they can be of different types.
- When multiple nominal values are specified in the fourth subfield, they must be separated by commas and be of the same type. The descriptive subfields apply to all the nominal values.

Note. Separate constants are generated for each separate nominal value specified.

No blanks are allowed in the DC instruction:

- Between subfields
- Between multiple operands
- Within any subfields, unless they occur as part of the nominal value of a C-, P-, or S-type character constant, or as part of a character self-defining term in a modifier expression or in the duplication factor subfield

Constants defined by the DC instruction are assembled into an object module at the location where the DC instruction is coded. The value of the symbol that names the DC instruction is the address of the leftmost byte of the constant. For example, the instruction

```
HEXCON   DC    XL6'AD'
```

causes the assembler to generate the 6-byte constant
0000 0000 00AD
and assign the address of the leftmost byte to the symbol HEXCON.

Length Attribute Value of Symbols Naming Constants

The length attribute value assigned to symbols in the name field of constants is equal to:

- The implicit length of the constant when no explicit length is specified in the operand of the constant or
- The explicitly specified length of the constant.

Note. If more than one operand is present, the length attribute value of the symbol is the length, in bytes, of the first constant, depending on its implicit or explicitly specified length.

The following table shows some sample DC instructions for the various constant types:

Type	*Implicit length (Bytes)*	*DC Instruction*	*Alignment (If explicit length not specified)*	*Value of length attribute*
B	as needed	DC B'101'	Byte	1
C	as needed	DC C'ABCD'	Byte	4
P	as needed	DC P'ABCD'	Byte	4
S	as needed	DC S'ABCD'	Byte	4
X	as needed	DC X'FFFF'	Byte	2
F	2	DC F'2'	Word	2
H	1	DC H'2'	Byte	1
D	4	DC D'2'	Word	4
E	4	DC E'1.414'	Word	4
L	8	DC L'1.414'	Word	8
A	2	DC A(TABLE)	Word	2
V	2	DC V(EXTDATA)	Word	2
W	2	DC W(WEAKDATA)	Word	2
N	2	DC N(DISK)	Word	2

Padding and Truncating Constants

The nominal values specified for constants are assembled into storage. The amount of space available for the nominal value of a constant is determined:

- By the explicit length specified in the second subfield of the operand, or
- If no explicit length is specified, by the implicit length according to the type of constant defined (see preceding table).

If more space is available than is needed to hold the binary representation of the nominal value, the extra space is padded:

- With binary zeros on the left for the binary (B), hexadecimal (X), fixed-point (F, H, and D), address (A) constants
- With binary zeros on the right for floating-point (E and L) constants
- With blank character codes on the right for the character (C, P, and S) constants

Note. Binary zeros are always assembled for N-, V- and W-type constants.

The following examples indicate the results of padding the different type data constants.

Constant definition	*Value assembled (Hex)*
DC B'101'	05
DC X'F'	0F
DC XL4'C4F'	00000C4F
DC F'255'	00FF
DC H'6'	06
DC D'202010'	0003151A
DC E'575E2'	44B09C00
DC L'73E4'	45B2390000000000
DC AL2(512)	0200
DC CL6'ABCD'	C1C2C3C44040
DC SL4'A'	41A0A0A0
DC P'ABC'	E2E4E781

If less space is available than is needed to hold the nominal value, the nominal value is truncated and part of the constant is lost or the value is assembled as zero. Truncation of the nominal value is:

- On the left for binary (B) and hexadecimal (X) constants.
- On the right for character (C, P, and S) constants.

The following types of constants are not truncated:

- Fixed-point (F, H, and D) constants are rounded if necessary. If the value exceeds the allowable range, zeros are assembled into the field.
- Floating-point (E and L) constants are rounded.
- Address (A) is not truncated. If the nominal value cannot be represented in the space available, the constant is flagged and assembled as zero.

The following examples indicate the results of truncating the different type data constants.

Constant definition	*Value assembled (Hex)*
DC BL2'01111000011110000'	F0F0
DC XL4'FFC00FFC8'	FC00FFC8
DC A(65535+1)	0000 (*ERROR*)
DC CL2'ABCD'	C1C2
DC SL2'ABC'	4142
DC F'32770'	0000 (*ERROR*)
DC F'1.2'	0001
DC H'–160'	00 (*ERROR*)

DC Operand Subfield 1: Duplication Factor

The duplication factor, if specified, causes the nominal value or multiple nominal values specified in a constant to be generated the number of times indicated by the factor. It is applied after the nominal value or values are assembled into one constant.

The factor can be specified by an unsigned decimal self-defining term or by an absolute expression enclosed in parentheses. The expression should have a positive value or be equal to zero. Any symbols used in the expression must have been previously defined.

The following examples indicate the results of specifying a duplication factor in the constant definition:

Constant definition	*Value assembled (Hex)*
DC 3F'240'	00F000F000F0
DC 2F'3,4'	0003000400030004
A EQU 5	
DC (A – 3) F'5'	00050005

Notes.

1. A duplication factor of zero is permitted with the following results:
   a. No value is assembled.
   b. If the zero duplication factor is used on an F-, D-, E-, L-, N-, A-, W-, or V-type constant, the location counter is forced to a word boundary. A byte of zeros is placed in the object text.
2. If duplication is specified for an address constant containing a location counter reference, the value of the location counter reference is not increased until after the DC instruction is completely processed. Therefore, it generates 2 words, each containing the address of X.

```
X  DC  2A(*)
```

DC Operand Subfield 2: Type

The type subfield must be coded. It defines the type of constant to be generated and is specified by a single letter code as shown below.

The type specification indicates to the assembler:

1. How the nominal value (or values) coded in subfield 4 is to be assembled; that is, which binary representation the object code of the constant must have.
2. How much storage the constant is to occupy, according to the implicit length of the constant if no explicit length specification is present. (For details see "Padding and Truncating Constants".)

Code	*Type of Constant*	*Machine formats*
C	EBCDIC	8-bit code for each character
S	ASCII	8-bit code for each character
P	PTTC/EBCD	8-bit code for each character
X	Hexadecimal	4-bit code for each digit
B	Binary	1-bit for each digit
F	Fixed-point	Signed, Fixed-point binary; normally 2 bytes (can be 1 2 bytes)
H	Fixed-point	Signed, Fixed-point binary; always 1 byte
D	Fixed-point	Signed, Fixed-point binary; normally 4 bytes (can be 1 4 bytes)
E	Floating-point	Floating-point binary; normally 4 bytes (can be 2 4 bytes)
L	Floating-point	Floating-point binary; normally 8 bytes (can be 2 8 bytes)
A	Address	Value of address or expression; 1 4 bytes
V	Address	Space reserved for external address; always one word
W	Address	Space reserved for external address; always one word
N	Name	Space reserved for resource reference value; always one word

```
* DC TYPE EXAMPLES                    OBJECT CODE IN HEXADECIMAL
        DC    C'AB'                   C1C2
        DC    S'ASCII'                4153 C3C9 C9Aø
        DC    X'FFFA'                 FFFA
        DC    B'ø1ø1'                 ø5
        DC    F'12.5'                 øøøD
        DC    H'-3'                   FD
        DC    D'123øø'                øøøø 3øøC
        DC    E'12.5'                 41C8 øøøø
        DC    L'-12.5'                C1C8 øøøø øøøø øøøø
        DC    A(ADDRESS)              (Value of location counter where ADDRESS is defined)
        DC    V(EXTERNAL)             øøøø
        DC    W(WEAK)                 øøøø
        DC    N(DISK)                 øøøø
```

DC Operand Subfield 3: Modifiers

The 3 modifiers you can code to describe a constant are:

- The length modifier (L), which explicitly defines the length in bytes desired for a constant.
- The scale modifier (S), which is only used with the fixed-point or floating-point constants. (For details, see "Scale Modifier".)
- The exponent modifier (E), which is only used with fixed-point or floating-point constants, and indicates the power of 10 by which the constant is to be multiplied before conversion to its internal binary format.

If multiple modifiers are used, they must appear in the sequence: length, scale, exponent.

Length Modifier

The length modifier indicates the explicit number of bytes into which the constant is to be assembled. You code it as Ln, where n is either of the following:

- A decimal self-defining term. For example:

```
SDTERM   DC    XL3'FF'
```

- An absolute expression enclosed in parentheses. It must have a positive value and any symbols it contains must have been previously defined; that is, in an instruction that physically precedes this DC in the source module. For example:

```
A    EQU   6
     :
     DC    XL(A+4)'FF'
```

When you specify the length modifier:

- Its value determines the number of bytes allocated to a constant. It therefore determines whether the nominal value of a constant must be padded or truncated to fit into the space allocated. (See "Padding and Truncating Constants".)
- Any boundary alignment normally implied by the constant type is lost. The constant assembles starting with the next available byte.
- Its value must not exceed the maximum length allowed. (For the allowable range of length modifiers, see the specifications for the individual constants and areas in this chapter.)

Scale Modifier

The scale modifier specifies the amount of scaling (shifting) desired for fixed-point or floating-point constants. The scale modifier specifies a shifting count of:

- Binary digits for fixed-point (F, H, and D) constants
- Hexadecimal digits for floating-point (E and L) constants

The scale modifier is written as Sn, where n is either:

- A decimal self-defining term or
- An absolute expression enclosed in parentheses. Any symbols used in the expression must have been previously defined; that is, in an instruction that physically precedes this DC in the source module.

Both types of specifications can be preceded by a plus or minus sign; if no sign is present, a plus sign is assumed.

```
SCALE    DC    FS3'2.25'
         :
PLUS3    DC    FS+3'2.25'
         :
MINUS2   DC    FS-2'141.4'
         :
         :
A        EQU   4
EXPRS    DC    FS(A*3)'1.8'
         :
         :
         :
FLTPT    DC    ES4'40.77'
         :
         :
```

Fixed-point; the allowable range for scale modifier is −31 to +63

Floating-point; the allowable range for scale modifier is 0 through 14

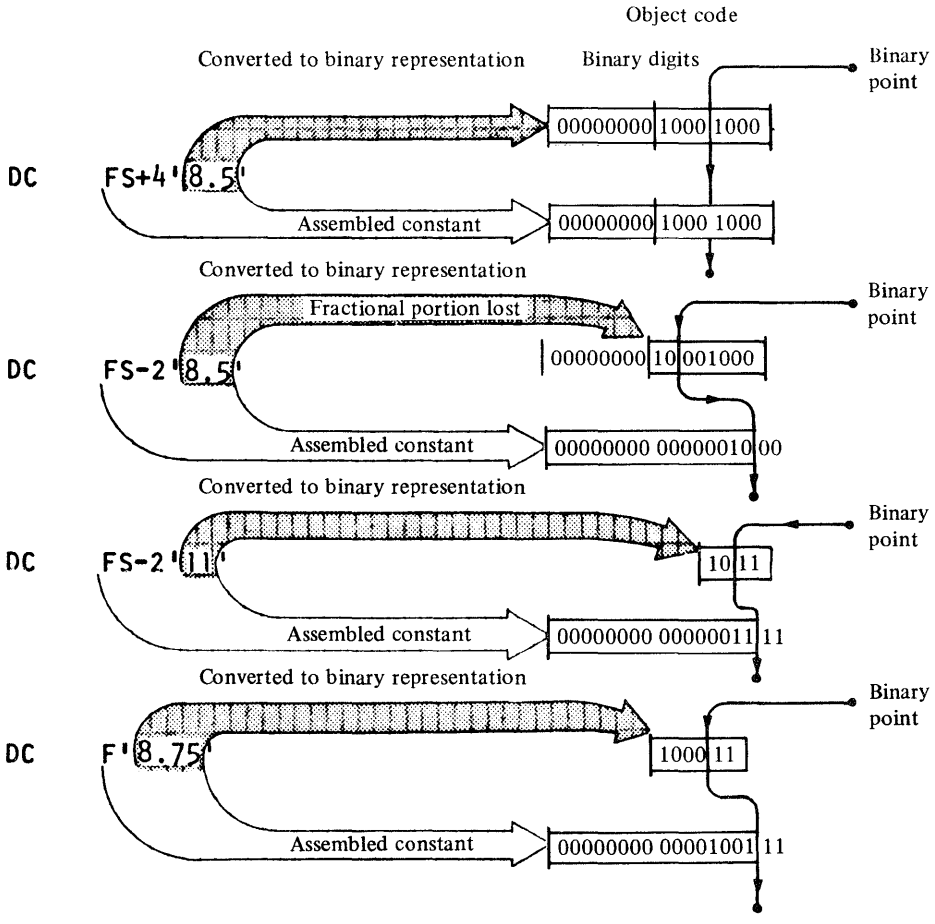
Scale Modifier for Fixed-point Constants. The scale modifier for fixed-point constants specifies the power of 2 by which the fixed-point constant is to be multiplied after its nominal value has been converted to a binary representation,

but before it is assembled in its final *scaled* form. Scaling causes the binary point to move from its assumed fixed position at the right of the rightmost bit position.

```
                                    Object code (binary)
DC      F'2'            00000000 00000010
:
DC      FS+2'2'         00000000 00001000
:
DC      FS+2'2.25'      00000000 00001001
:
DC      FS-2'8'         00000000 00000010
```

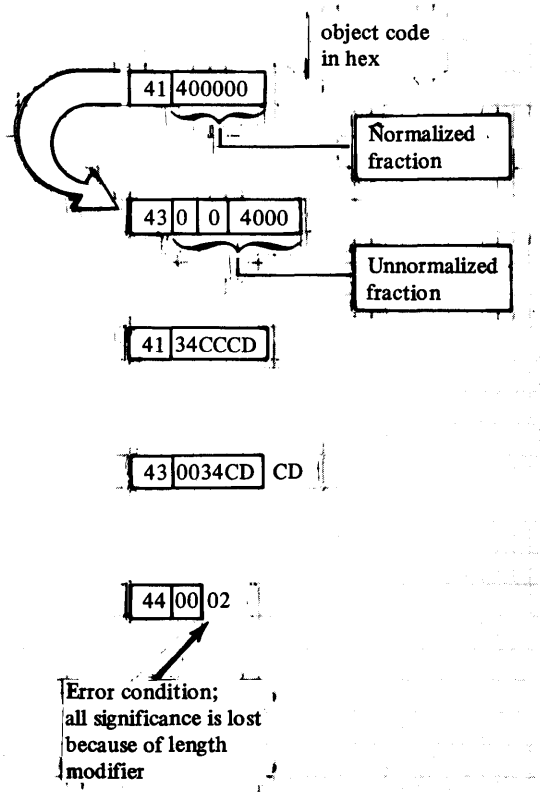
Notes.

1. When the scale modifier has a positive value, it indicates the number of binary positions to be occupied by the fractional portion of the binary number.
2. When the scale modifier has a negative value, it indicates the number of binary positions to be deleted from the integer portion of the binary number.
3. When positions are lost because of scaling (or lack of scaling), rounding occurs in the leftmost bit of the lost portion. The rounding is reflected in the rightmost position retained.
4. Scaling must not cause a value overflow condition, nor is it permitted that all significant bits be lost.
5. The assembler must be able to internally maintain the fixed point number prior to scaling. The integer portion must be represented in 32 bits.



Scale Modifier for Floating-point Constants. The scale modifier for floating-point constants must have a positive value. It specifies the number of hexadecimal positions that the fractional portion (mantissa) of the binary representation of a floating-point constant is to be shifted to the right. The hexadecimal point is assumed to be fixed at the left of the leftmost position in the fractional field. When scaling is specified, it causes an unnormalized hexadecimal fraction to be assembled. (A number is unnormalized when the leftmost positions of the fraction contain hexadecimal zeros). The magnitude of the constant is retained because the exponent in the characteristic portion of the constant is adjusted upward accordingly. When hexadecimal positions are lost, rounding occurs in the leftmost hexadecimal position of the lost portion. The rounding is reflected in the rightmost position retained. Scaling must not cause all significant mantissa digits to be lost.

```
         DC    E'4'
         DC    ES2'4'
         DC    E'3.3'
ROUND    DC    ES2'3.3'
ERROR    DC    EL2S3'2'
```



Exponent Modifier

The exponent modifier specifies the power of 10 by which the nominal value of a constant is multiplied before being converted to its internal binary representation. You can only use it with the fixed-point (F, H, and D) and floating-point (E and L) constants. You code the exponent modifier as En, where n is either:

- A decimal self-defining term or
- An absolute expression enclosed in parentheses. Any symbols used in the expression must be previously defined, that is, in an instruction that physically precedes this DC in the source module.

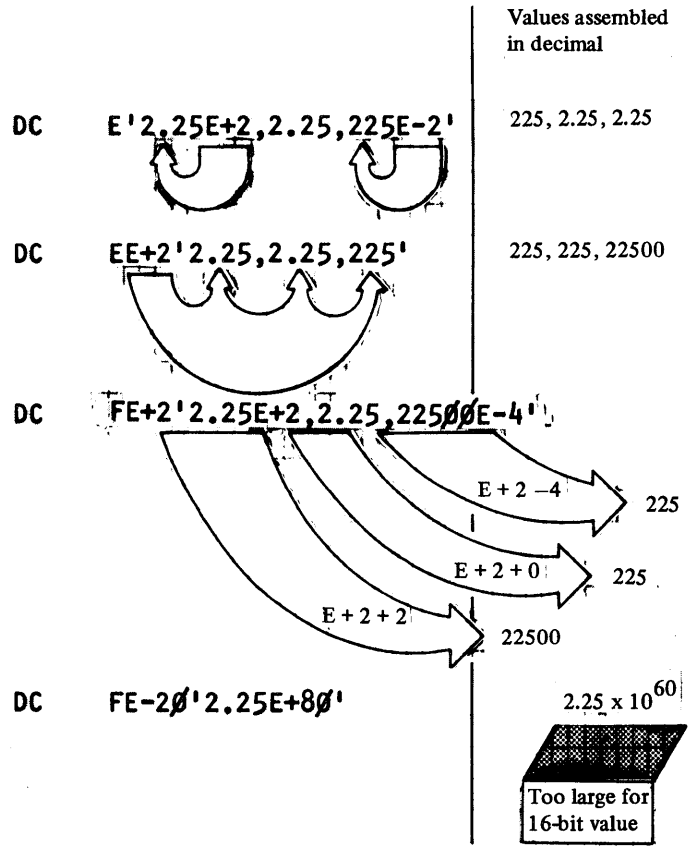
You can precede the decimal self-defining term or the expression with a sign; if no sign is present, a plus sign is assumed. The range for the exponent modifier is −85 through +75.

```
A    EQU   5
B    EQU   1
     DC    F'4'
     :
     DC    FE2'4'
     :
     DC    FE(A-B*3)'4'
     :
     DC    FE-2'4ØØ'
```

Decimal value before conversion to binary form	Object code (Binary digits)
4	00000000 00000100
400	00000001 10010000
400	00000001 10010000
4	00000000 00000100

Notes.

1. Do not confuse the exponent modifier with the exponent you specify in the nominal value subfield of fixed-point and floating-point constants. The exponent modifier affects each nominal value in the operand, whereas the exponent you code as part of the nominal value subfield affects only that nominal value. If both types of exponent specification are present in a DC operand, their values are algebraically added together before the nominal value is converted to binary form. However, this sum must be within the permissible range of −85 through +75.
2. The value of the constant, after exponents are applied, must be contained in the implicit or explicitly specified length of the constant. Also, significance must not be permitted to be completely lost.



Storage Requirements for Constants. The total amount of storage required to assemble a DC instruction operand is any bytes skipped for alignment, plus the product of:

- The length (implicit or explicit), and
- The duplication factor (if specified)

The maximum amount of storage allowed for a constant is 65535 bytes.

DC Operand Subfield 4: Nominal Value

You must code the nominal value subfield in a DC instruction. It defines the value of the constant (or constants) described and affected by the subfields that precede it. It is this value that assembles into the internal binary representation of the constant.

Only one nominal value is allowed for C-, S-, P-, B-, and X-type constants. How nominal values are specified and interpreted by the assembler is explained in the sections that describe each individual constant. The formats for specifying nominal values are described in the following table.

	Format of nominal value subfields	
Constant type	*Single*	*Multiple*
C P S B X	'value'	Not allowed
F H D E L	'value'	'value,value,. . ., value' *separated by commas*
A V W N	(value)	(value,value, . . . , value)

EBCDIC Character Constant (C)

This character constant specifies character strings that the assembler converts into their internal EBCDIC representation.

The maximum number of bytes generated by one DC statement is 65535. Each character specified in the nominal value subfield assembles into one byte.

Multiple nominal values are not allowed; the assembler considers the comma as a valid string character. Scale and exponent modifiers are not allowed.

Note. When ampersands or apostrophes are to be included in the assembled constant, double ampersands or double apostrophes must be specified; they are assembled as a single ampersand or single apostrophe.

The contents of the subfields defining a character constant are described by the following examples.

```
* DC INSTRUCTION                DESCRIPTION
*
DUP       DC    4ØC' '         DUPLICATION FACTOR IS ALLOWED
          :
LENGTH    DC    C'ABCD'        IMPLICIT LENGTH AS NEEDED
          :
RANGE     DC    CL256' '       RANGE FOR LENGTH IS 1 to 256 BYTES
          :
VALUE     DC    C'AB12#$'      VALUE REPRESENTED BY CHARACTERS
          DC    C''''          TWO APOSTROPHES ASSEMBLE AS ONE
          DC    C'&&'          TWO AMPERSANDS ASSEMBLE AS ONE
          DC    C','           COMMA ASSEMBLES AS COMMA
          :
ENCLOS    DC    C'   '         NOMINAL VALUE ENCLOSED BY
*                                APOSTROPHES
          :
MULTI     DC    C'AB,C'        MULTIPLE NOMINAL VALUES NOT POSSIBLE
          :
PAD       DC    CL2'A'         PADDED WITH EBCDIC BLANKS AT RIGHT
          :
TRUNC     DC    CL1'ABC'       TRUNCATION OF VALUE AT RIGHT
```

ASCII Character Constant (S)

This character constant specifies character strings, such as message text, that the assembler converts into their internal ASCII representation. ASCII code is generated as a 7-bit character code with the high-order bit of zero.

The maximum number of bytes generated by one DC statement is 65535. Each character specified in the nominal value subfield assembles into one byte.

Multiple nominal values are not allowed; the assembler considers the comma as a valid string character. Scale and exponent modifiers are not allowed.

Note. Specify double ampersands or double apostrophes for each single ampersand or single apostrophe you want assembled into the constant.

```
* DC INSTRUCTION                DESCRIPTION
*
DUP       DC    4ØS' '         DUPLICATION FACTOR IS ALLOWED
          :
LENGTH    DC    S'ABCD'        IMPLICIT LENGTH AS NEEDED
          :
RANGE     DC    SL256' '       RANGE FOR LENGTH IS 1 to 256 BYTES
          :
VALUE     DC    S'AB12#$'      VALUE REPRESENTED BY CHARACTERS
          DC    S''''          TWO APOSTROPHES ASSEMBLE AS ONE
          DC    S'&&'          TWO AMPERSANDS ASSEMBLE AS ONE
          DC    S','           COMMA ASSEMBLES AS COMMA
          :
ENCLOS    DC    S'   '         NOMINAL VALUE ENCLOSED BY
*                                APOSTROPHES
          :
MULTI     DC    S'AB,C'        MULTIPLE NOMINAL VALUES NOT POSSIBLE
          :
PAD       DC    SL2'A'         PADDED WITH ASCII BLANKS AT RIGHT
          :
TRUNC     DC    SL1'ABC'       TRUNCATION OF VALUE AT RIGHT
```

PTTC/EBCD Character Constant (P)

This character constant specifies character strings that the assembler converts into their internal PTTC/EBCD representation.

The maximum number of bytes generated by one DC statement is 65535. Each character you code in the nominal value subfield assembles into one byte.

Multiple nominal values are not allowed; the assembler considers the comma as a valid string character. Scale and exponent modifiers are not allowed.

Note. Specify double apostrophes or ampersands for each single apostrophe or ampersand you want assembled into the constant.

```
DUP      DC    4ØP' '              DUPLICATION FACTOR IS ALLOWED.

LENGTH   DC    P'ABCD'             IMPLICIT LENGTH AS NEEDED.

RANGE    DC    PL256' '            RANGE FOR LENGTH IS 1 THROUGH 256
*                                  BYTES.

* SCALE MODIFIER AND EXPONENT MODIFIER ARE NOT ALLOWED.

VALUE    DC    P'AB12#$'           VALUE REPRESENTED BY CHARACTERS.
         DC    P''''               2 APOSTROPHES ASSEMBLE AS ONE.
         DC    P'&&'               2 AMPERSANDS ASSEMBLE AS ONE.
         DC    P','                COMMA ASSEMBLES AS COMMA.

ENCLOS   DC    P' '                NOMINAL VALUE ENCLOSED BY
*                                  APOSTROPHES.

* EXPONENT NOT ALLOWED IN NOMINAL VALUE SUBFIELD.

MULTI    DC    P'AB,C'             MULTIPLE NOMINAL VALUES NOT
*                                  POSSIBLE.

PAD      DC    P'A'                PADDED WITH PTTC/EBCD BLANKS AT
*                                  RIGHT.

TRUNC    DC    PL1'ABC'            TRUNCATION OF VALUE OF RIGHT.
```

Hexadecimal Constant (X)

You can use hexadecimal constants to generate large bit patterns more conveniently than with binary constants. Also, the hexadecimal values you specify in a source module allow you to compare them directly with the hexadecimal values generated for the object code and address locations printed in the program listing.

Each hexadecimal digit specified in the nominal value subfield is assembled into 4 bits. The implicit length in bytes of a hexadecimal constant is then one-half the number of hexadecimal digits specified (assuming that the number of digits is a multiple of 2).

The contents of the subfields defining a hexadecimal constant are described by the following examples.

```
* DC INSTRUCTION                    DESCRIPTION
*
DUP      DC    2X'Ø'               DUPLICATION FACTOR IS ALLOWED
         :
LENGTH   DC    X'FFFF'             IMPLICIT LENGTH AS NEEDED
         :
RANGE    DC    XL256'Ø'            RANGE FOR LENGTH IS 1 to 256 BYTES
         :
VALUE    DC    X'Ø9AF'             VALUE REPRESENTED BY
*                                  HEXADECIMAL DIGITS
         :
ENCLOS   DC    X'FFFF'             NOMINAL VALUE ENCLOSED BY
*                                  APOSTROPHES
         :
MULTI    DC    X'AB,C'             MULTIPLE NOMINAL VALUES NOT POSSIBLE
         :
PAD      DC    X'F'                PADDED WITH BINARY ZEROS AT LEFT
         :
TRUNC    DC    XL1'FFFFF'          TRUNCATION OF VALUE AT LEFT
```

Binary Constant (B)

The binary constant specifies the precise bit pattern you want to assemble into storage. Each binary constant assembles into the integral number of bytes required to contain the bits specified.

The contents of the subfields defining a binary constant are described by the following examples.

```
* DC INSTRUCTION                    DESCRIPTION
*
DUP      DC    2B'1Ø1'             DUPLICATION FACTOR IS ALLOWED
         :
LENGTH   DC    B'1'                IMPLICIT LENGTH AS NEEDED
         :
RANGE    DC    BL256'Ø'            RANGE FOR LENGTH IS 1 to 256 BYTES
         :
VALUE    DC    B'1Ø1Ø1Ø'           VALUE REPRESENTED BY BINARY DIGITS
         :
ENCLOS   DC    B'Ø'                NOMINAL VALUE ENCLOSED BY
*                                  APOSTROPHES
         :
MULTI    DC    B'1Ø,1'             MULTIPLE NOMINAL VALUES NOT POSSIBLE
         :
PAD      DC    B'11ØØ1'            PADDED WITH BINARY ZEROS AT LEFT
         :
TRUNC    DC    BL1'1111ØØØØ1111ØØØØ1'   TRUNCATION OF VALUE AT LEFT
```

Fixed-point Constant (F)

A fixed-point constant is written as a decimal number and can be followed by a decimal exponent. The number can be an integer, a fraction, or a mixed number (one with integral and fractional portions). The format of the constant is as follows:

- The number is written as a signed or unsigned decimal value. The decimal point can be placed before, within, or after the number. If it is omitted, the number is assumed to be an integer. A positive sign is assumed if an unsigned number is specified.
- The exponent is optional. If specified, it is written immediately after the number as En, where n is an optionally signed decimal value specifying the exponent of the factor 10. The exponent must be in the range −85 to +75. If an unsigned exponent is specified, a plus sign is assumed.

The number is converted to binary, the exponent and scale factor (if any) are applied, the number is rounded and assembled into the proper field, according to the specified or implied length. An implied length of 2 bytes is assumed if a length is not specified. The resulting number does not differ from the exact value by more than one in the last binary position. If the value of the number exceeds the allowable range (−32768 to 32767), the statement is flagged and a zero is assembled into the whole field. Any duplication factor that is present is applied after the constant is assembled. A negative number is carried in twos complement form.

```
* DC INSTRUCTION                    DESCRIPTION
*
DUP       DC    4F'1'               DUPLICATION FACTOR IS ALLOWED
          :
LENGTH    DC    F'1ØØ'              IMPLICIT LENGTH IS ALWAYS 2 BYTES
          :
RANGE     DC    FL1'1ØØ'            LENGTH MODIFIER MUST BE 1 OR 2,
*                                   IF USED
          :
VALUE     DC    F'32767'            VALUE REPRESENTED BY DECIMAL DIGITS
          :
ENCLOSE   DC    F'-1'               NOMINAL VALUE ENCLOSED BY
*                                   APOSTROPHES
          :
EXPVAL    DC    F'1.414E2'          EXPONENT ALLOWED IN NOMINAL VALUE;
*                                   RANGE FOR EXPONENT IS -85 TO +75
          :
PAD       DC    F'2Ø'               PADDED WITH BINARY ZEROS AT LEFT
          :
MULTI     DC    F'1,2,3'            MULTIPLE NOMINAL VALUES ALLOWED
          :
SCALE     DC    FS6'-25.46'         RANGE FOR SCALE IS -187 TO +346
          :
EXPON     DC    FE2'46415'          RANGE FOR EXPONENT IS -85 TO +75
```

Note. Truncation of F-type constants is not allowed.

Fixed-Point Constant (H)

An H-type (halfword) fixed-point constant is identical to an F-type constant, except that the H-type constant is assembled as a 1-byte field on a byte boundary. The maximum range of an H-type constant is −128 to 127.

```
* DC INSTRUCTION                    DESCRIPTION
*
DUP       DC    4H'1'               DUPLICATION FACTOR IS ALLOWED
          :
LENGTH    DC    H'10'               IMPLICIT LENGTH IS ALWAYS 1 BYTE
          :
RANGE     DC    HL1'10'             LENGTH MODIFIER MUST BE 1, IF USED
          :
VALUE     DC    H'35'               VALUE REPRESENTED BY DECIMAL DIGITS
          :
ENCLOS    DC    H'-1'               NOMINAL VALUE ENCLOSED BY
*                                   APOSTROPHES
          :
EXPVAL    DC    H'21E-1'            EXPONENT ALLOWED IN NOMINAL VALUES;
*                                   RANGE FOR EXPONENT IS -85 to +75
          :
PAD       DC    H'2'                PADDED WITH BINARY ZEROS AT LEFT
```

Note. Truncation of H-type constants is not allowed.

Fixed-Point Constant (D)

A D-type (doubleword) fixed-point constant is identical to an F-type constant, except that the D-type constant is assembled as a 4-byte field on a word boundary. The maximum range of a D-type constant is -2^{31} to $2^{31}-1$.

```
* DC INSTRUCTION                    DESCRIPTION
*
DUP       DC    4D'1'               DUPLICATION FACTOR IS ALLOWED
          :
LENGTH    DC    D'10000'            IMPLICIT LENGTH IS ALWAYS 4 BYTES
          :
RANGE     DC    DL4'10000'          LENGTH MODIFIER MUST BE 1, 2, 3 or
*                                   4, IF USED
          :
VALUE     DC    D'66'               VALUE REPRESENTED BY DECIMAL DIGITS
          :
ENCLOS    DC    D'-1'               NOMINAL VALUE ENCLOSED BY
*                                   APOSTROPHES
          :
EXPVAL    DC    D'231.62E-2'        EXPONENT ALLOWED IN NOMINAL VALUE;
*                                   RANGE FOR EXPONENT IS -85 to +75
          :
PAD       DC    D'100'              PADDED WITH BINARY ZEROS AT LEFT
```

Note. Truncation of D-type constants is not allowed.

Floating-Point Constant (E)

An E-type (single-precision) floating-point constant is written as a decimal number and can be followed by a decimal exponent. The number can be an integer, a fraction, or a mixed number (one with integral and fractional portions). The format of the constant is as follows:

- The number is written as a signed or unsigned decimal value. The decimal point can be placed before, within, or after the number. If it is omitted, the number is assumed to be an integer. A positive sign is assumed if an unsigned number is specified.
- The exponent is optional. If specified, it is written immediately after the number as En, where n is an optionally signed decimal value specifying the exponent of the factor 10. The exponent must be in the range −85 to +75. If an unsigned exponent is specified, a plus sign is assumed.

The external format for a floating-point number has 2 parts: the portion containing the exponent, which is called the characteristic, followed by the portion containing the fraction, which is called the mantissa. Therefore, the number specified as a floating-point constant must be converted to a fraction before it can be translated into the proper format.

For example, the constant 27.35E2 represents the number 27.35 times 10^2. Represented as a fraction, 27.35E2 would be 0.2735 times 10^4, the exponent having been modified to reflect the shifting of the decimal point. Thus, the exponent is also altered before being translated into machine format.

In machine format, a floating-point number also has 2 parts, the signed exponent and signed fraction. The quantity expressed by this number is the product of the fraction and the number 16 raised to the power of the exponent.

The exponent is translated into its binary equivalent in excess 64 binary notation and the fraction is converted to a binary number. Leading hexadecimal zeros are removed. Rounding of the fraction is then performed according to the specified or implied length, and the number is stored in the proper field. The resulting number does not differ from the exact decimal value by more than one in the last place.

The maximum range of the magnitude of an E-type constant is approximately 10^{-78} to 10^{76}. If this range is exceeded, the DC instruction is flagged and a zero is assembled into the whole field.

Within the portion of the floating-point field allocated to the fraction, the hexadecimal point is assumed to be to the left of the leftmost hexadecimal digit, and the fraction occupies the leftmost portion of the field. Negative fractions are carried in true representation, not in the twos complement form.

As an example, the machine representation of the floating-point constant E'55.125' would be:

Bit	*Portion of constant*	*Contents*
0	Sign bit of mantissa	0
1–7	Exponent	X'42'
8–31	Mantissa	X'372000'

```
* DC INSTRUCTION                    DESCRIPTION
*
DUP       DC    4E'.ØØØ25'         DUPLICATION FACTOR IS ALLOWED
          :
LENGTH    DC    E'1.315'           IMPLICIT LENGTH IS ALWAYS 4 BYTES
          :
RANGE     DC    EL4'1.111'         LENGTH MODIFIER MUST BE 2, 3, OR 4,
*                                  IF USED
          :
VALUE     DC    E'1.41416'         VALUE REPRESENTED BY DECIMAL DIGITS
          :
ENCLOS    DC    E'1'               NOMINAL VALUE ENCLOSED BY
*                                  APOSTROPHES
          :
EXPVAL    DC    E'1E-8Ø'           EXPONENT ALLOWED IN NOMINAL VALUE;
*                                  RANGE FOR EXPONENT IS -85 TO +75
          :
PAD       DC    E'5Ø'              PADDED WITH BINARY ZEROS AT RIGHT
          :
TRUNC     DC    E'-123.456789'     VALUE IS ROUNDED
          :
SCALE     DC    ES6'1ØØ'           RANGE FOR SCALE IS Ø THROUGH 6
          :
EXPON     DC    EE-85'1'           RANGE FOR EXPONENT IS -85 TO +75
```

Floating-Point Constant (L)

An L-type (double-precision) floating point constant is identical to an E-type constant except that the L-type constant is assembled as an 8-byte field on a word boundary. The resulting constant consists of a 1-byte sign and exponent plus a 7-byte hexadecimal fraction.

```
* DC INSTRUCTION                    DESCRIPTION
*
DUP       DC    4L'.ØØ25'          DUPLICATION FACTOR IS ALLOWED
          :
LENGTH    DC    L'1.315'           IMPLICIT LENGTH IS ALWAYS 8 BYTES
          :
RANGE     DC    LL8'1.315'         LENGTH MODIFIER MUST BE 2, 3, 4, 5,
*                                  6, 7, OR 8, IF USED
          :
VALUE     DC    L'1.41416'         VALUE REPRESENTED BY DECIMAL DIGITS
          :
ENCLOS    DC    L'1'               NOMINAL VALUE ENCLOSED BY
*                                  APOSTROPHES
          :
EXPVAL    DC    L'1E-8Ø'           EXPONENT ALLOWED IN NOMINAL VALUE;
*                                  RANGE FOR EXPONENT IS -85 TO +75
          :
PAD       DC    L'5Ø'              PADDED WITH BINARY ZEROS AT RIGHT
```

A-Type Address Constant

This section and the two following sections describe how the different types of address constants assemble from expressions that usually represent storage addresses, and how you use the constants for addressing within and between source modules.

In the A-type address constant, you can specify any of the three types of assembly-time expressions (see "Expressions" in Chapter 2), whose value the assembler then computes and assembles into object code. You use this expression computation as follows:

- Relocatable expressions for addressing
- Absolute expressions for addressing and value computation
- Complex relocatable expressions to relate addresses in different source modules.

The value of the location counter reference (*) when specified in an address constant does not vary from constant to constant if a duplication factor, multiple nominal values, or multiple operands are specified.

The contents of the subfields defining the A-type address constants are described by the following examples.

```
* DC INSTRUCTION                    DESCRIPTION
*
DUP       DC    4A(*)               DUPLICATION FACTOR ALLOWED
          :
LENGTH    DC    A(LABEL)            IMPLICIT LENGTH IS ALWAYS 2 BYTES
          :
RANGE     DC    AL1(LABEL)          LENGTH MODIFIER CAN BE FROM 1 TO 4;
*                                   ONLY LENGTH 2 IS VALID FOR A
*                                   RELOCATABLE VALUE--LENGTHS 1, 3, AND
*                                   4 MUST BE FOR ABSOLUTE VALUES
          :
VALUE     DC    A(LABEL+2)          VALUE REPRESENTED BY ANY EXPRESSION
          :
ENCLOS    DC    A(*-*)              NOMINAL VALUE ENCLOSED BY
*                                   PARENTHESES
          :
MULTI     DC    A(LABEL,SYMBOL)     MULTIPLE NOMINAL VALUES ALLOWED
          :
PAD       DC    A(1)                PADDED WITH BINARY ZEROS AT LEFT
```

Note. Truncation of A-type constants is not allowed; if the value is too large, a zero is assembled and the statement is flagged as an error.

V-Type Address Constant

The V-type address constant reserves storage for the address of a location in another module. You can use the V-type address constant to branch to the external address. (There are other ways to branch to external addresses, as described in "Symbolic Addressing Between Source Modules—Symbolic Linkage" later in this chapter.)

When you specify a symbol in a V-type address constant, the assembler assumes that it is an external symbol. A value of zero is assembled into the space reserved for the V-type constant; the correct relocated value of the address is inserted into this space by the application builder. The symbol specified in the nominal value subfield does not constitute a definition of the symbol for the source module in which the V-type address constant appears.

The contents of the subfields defining the V-type address constants are described in the following examples.

```
* DC INSTRUCTION                    DESCRIPTION
*
DUP       DC    4V(EXTERNAL)       DUPLICATION FACTOR IS ALLOWED
          :
LENGTH    DC    V(EXTERNAL)        IMPLICIT LENGTH IS ALWAYS 2 BYTES
          :
RANGE     DC    VL2(EXTERNAL)      LENGTH MODIFIER MUST BE 2, IF USED
          :
VALUE     DC    V(EXTERNAL)        VALUE REPRESENTED BY SINGLE
*                                  RELOCTABLE SYMBOL
          :
ENCLOS    DC    V(EXTERNAL)        NOMINAL VALUE ENCLOSED BY
*                                  PARENTHESES
          :
MULTI     DC    V(EXT1,EXT2)       MULTIPLE NOMINAL VALUES ALLOWED
```

Note. Truncation of V-type constants is not applicable.

W-Type Address Constant

Specified as one relocatable symbol, the W-type address constant reserves storage for the address of a weak external symbol that refers to other modules. The automatic library call mechanism (AUTOCALL) of the application builder is not activated for symbols identified by a weak external reference. The implied length of a W-type address constant is 2 bytes. Specifying a symbol as the operand of the constant does not constitute a definition of the symbol.

```
* DC INSTRUCTION                    DESCRIPTION
*
DUP       DC    4W(WEAK)           DUPLICATION FACTOR IS ALLOWED
          :
LENGTH    DC    W(WEAK)            IMPLICIT LENGTH IS ALWAYS 2 BYTES
          :
RANGE     DC    WL2(WEAK)          LENGTH MODIFIER MUST BE 2, IF USED
          :
VALUE     DC    W(WEAK)            VALUE REPRESENTED BY SINGLE
*                                  RELOCATABLE SYMBOL
          :
ENCLOS    DC    W(WEAK)            NOMINAL VALUE ENCLOSED BY
*                                  PARENTHESES
          :
MULTI     DC    W(WXT1,WXT2)       MULTIPLE NOMINAL VALUES ALLOWED
```

Note. Truncation of W-type constants is not applicable.

N-Type Name Constant

Specified as one symbol, the N-type constant reserves storage for the value of the resource reference constant to be resolved by the application builder.

A resource reference constant must be used when two or more programs reference, through supervisor assistance, a specific name (as opposed to a storage address).

The implied length of the N-type constant is two bytes. Specifying a symbol as the operand of the constant does not constitute a definition of the symbol.

Example:

```
DC INSTRUCTION                          DESCRIPTION

DUP       DC    4N(DISK)                DUPLICATION FACTOR IS ALLOWED
LENGTH    DC    N(DISK)                 IMPLICIT LENGTH IS ALWAYS 2 BYTES
RANGE     DC    NL2(DISK)               LENGTH MODIFIER MUST BE 2, IF USED
VALUE     DC    N(DISK)                 VALUE REPRESENTED BY A SINGLE
                                        SYMBOL
ENCLOS    DC    N(DISK)                 NOMINAL VALUE ENCLOSED BY
                                        PARENTHESES
MULTI     DC    N(DISK1,DISK2)          MULTIPLE NOMINAL VALUES ALLOWED
```

The DS Instruction

The DS instruction allows you to:

- Reserve areas of storage
- Provide labels for these areas
- Use these areas by referring to the symbols defined as labels

The format of the DS instruction is like that of the DC instruction:

Name	*Operation*	*Operand*
[label]	DS	[dup] type[mods] [{'value' / (value)}] [,opnd 2]...

where operand consists of the same 4 subfields as the DC statement. However, with the DS instruction no data is assembled and the nominal value subfield is therefore optional.

The subfields for the DS instruction operand are:

Subfield	*Contents*
1	duplication factor (optional)
2	data type (required)
3	modifiers (optional)
4	nominal value or values (optional)

The maximum length that can be specified in a DS operand is 65535 bytes.

The label of a DS instruction, like the label of a DC instruction, has an address value of the leftmost byte of the area reserved.

If the DS instruction is specified with more than one nominal value, the label addresses the area reserved for the field that corresponds to the first nominal value.

Using the DS instruction to reserve storage. The DS instruction is the best way to symbolically define storage for work areas and I/O buffers. If you wish to take advantage of implicit length calculation, do not supply a length modifier in your operand specification. Specify a type subfield that corresponds to the type of area you need.

```
* DS INSTRUCTION                    STORAGE RESERVED
*
FAREA    DS    F                    2 BYTES
XAREA    DS    X                    2 BYTES
DUPFAC   DS    8F                   16 BYTES
EAREA    DS    3E                   12 BYTES
```

To reserve large areas, you can use a duplication factor. You can also use character (C and S) or hexadecimal (X) field types to specify large areas using the length modifier.

```
* DS INSTRUCTION                    STORAGE RESERVED
*
CAREA    DS    CL80                 80 BYTES
SAREA    DS    SL20                 20 BYTES
COMBIN   DS    64XL8                512 BYTES
```

Although the nominal value is optional for a DS instruction, you can put it to good use by letting the assembler compute the length of areas for the B-, C-, S-, and X-type areas. You achieve this by specifying the general format of the data that will be placed in the area at execution time.

You can force the location counter to a word boundary by using the appropriate data type with a duplication factor of zero. This method ensures a boundary alignment that you would otherwise not have.

```
* DS INSTRUCTION                    STORAGE RESERVED
*
CHAR     DS    C'MESSAGE TEXT'      12 BYTES
HEXONE   DS    X'F'                 1 BYTE
HEXTWO   DS    30X'F1F2'            60 BYTES
```

Using the DS instruction to name fields of an area. Using a duplication factor of zero in a DS instruction allows you to provide a label for an area of storage without actually reserving the area. You can use DS or DC instructions to reserve storage for and assign labels to fields within the area. These fields can then be addressed symbolically. (You can also do this with DSECTs as described later in this chapter.)

Nothing is assembled into the storage area reserved by a DS instruction. No assumption should be made as to the initial contents of the reserved area at execution time. In addition, no RLDs are generated for A, V, W, or N-type DS statements.

The size of a storage area that can be reserved by a DS instruction is limited by the maximum value of the location counter (65535).

Parameter Reference (PREF)

Parameter lists often need to be generated to pass information to routines. The PREF (parameter reference) instruction lets you generate a one to five word parameter list. PREF instructions are generated by IBM-supplied macros to create parameter lists. The locations of the parameters are resolved by the program manager.

The format of the PREF instruction is:

Name	*Operation*	*Operand*
[label]	PREF	Zero to four address specifications separated by commas

The PREF operand may contain up to four address specifications separated by commas. A specification may be omitted by coding two successive commas or by omitting trailing parameters.

Valid specifications are:

```
expression
expression*
(reg,disp)
(reg,disp)*
(reg)
(reg)*
reg
```

The *expression* may be any relocatable expression or an explicitly declared external symbol. Also, the expression may be in the range of a USING instruction. If the PREF instruction is in the domain of that same USING instruction, the assembler will convert the address to a *(reg,disp)* format. The *reg* must be specified as a valid register expression. The *disp* may be a self-defining term or an absolute expression, having a value in the range 0 to 4095.

Note. R0 may not be specified as any of the register type address specifications.

The PREF can generate up to five words. The first word contains four four-bit fields which indicate the address specification type for each parameter. The remaining words (up to four) contain the parameter address specifications.

The address specification types are:

Type (binary)	Address Specification
0000	omitted
0001	expression
0010	(reg) (reg, disp) expression with USING
0011	reg
0100	***invalid***
0101	expression*
0110	(reg)* (reg, disp)* expression* with USING

The parameter words contain:

Parameter word	Address specification
Value of expression 0 for external symbol	expression expression*
Register value in bits 0–3 Bits 4–15 are zero	reg
Register value in bits 0–3 Displacement (unsigned in bits 4–15)	(reg, disp) (reg, disp)* expression } with USING expression* } with USING

Alignment is to a word boundary.

Example:

```
PREF    BUF, , (R2)* , BUF2          Second parameter omitted
```

Note. If an error is detected during the processing of a PREF operand, the address specification type in the parameter word will be set to B'0100' (invalid). The address specification itself will be assembled as binary zeros.

Program Sectioning

This section explains how you can subdivide a large program into smaller parts so that they are easier to understand and maintain. It also shows how you can divide these smaller parts into convenient sections; for example, one section to contain your executable instructions and another section to contain your data constants and areas.

You should consider two distinct subdivisions when writing an assembler language program:

- Dividing the program into source modules
- Dividing the program into control sections

You can divide a program into two or more source modules. Each source module is assembled into a separate object module. You then use the application builder to combine the object modules into a load module, forming an executable program.

You can also divide a source module into two or more control sections. Each control section of a full assembly is assembled as part of the object module. The application builder processes these control sections, producing a load module with contiguous storage addresses.

Communication Between Program Parts

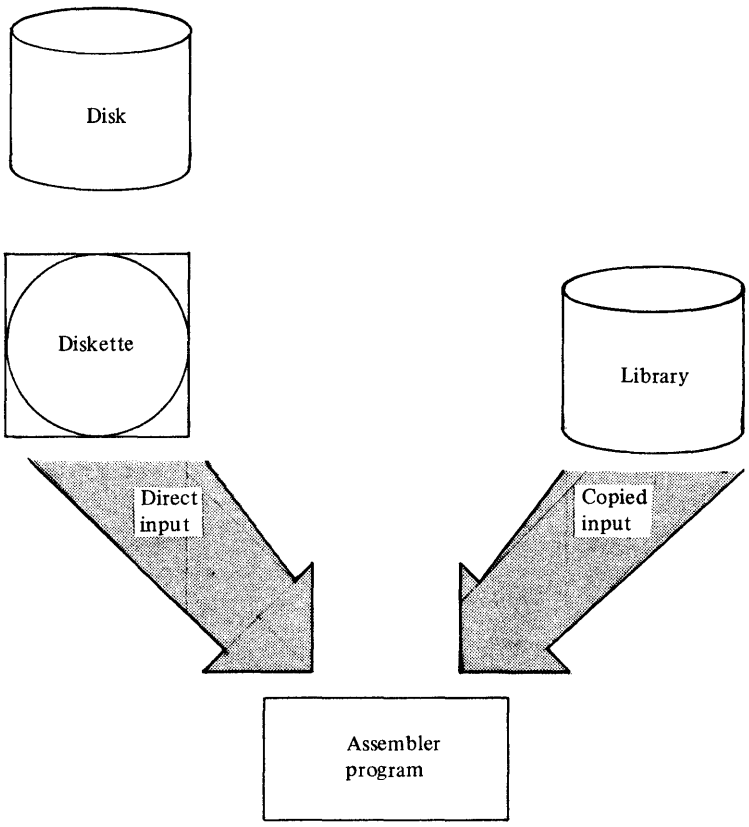
You must be able to communicate between the parts of your program; that is, be able to refer to data in a different part or be able to branch to another part:

- To communicate between 2 or more source modules, you must symbolically link them together; symbolic linkage is described in "Symbolic Addressing Between Source Modules—Symbolic Linkage" in this chapter.
- To communicate between control sections within a source module, you must establish the addressability of each control section; establishing addressability is described in "Symbolic Addressing Within Source Modules—Establishing Addressability" in this chapter.

The Source Module

A source module is composed of source statements in the assembler language. You can include these statements in the source module in 2 ways:

1. You write them on a coding form and then enter them as input; for example, using the text editor.
2. You specify one or more COPY instructions among the source statements being entered. When the assembler encounters a COPY instruction, it inserts a predetermined set of source statements from a library. These statements then become a part of the source module.



The Beginning of a Source Module

The first statement of a source module can be any assembler language statement described in this manual (except MEND or MEXIT). You should initiate the first control section of a source module with the START or CSECT instruction. However, you can, or in some cases must, write source statements before the beginning of the first control section. (For a list of these statements see "First Control Section" in this chapter.)

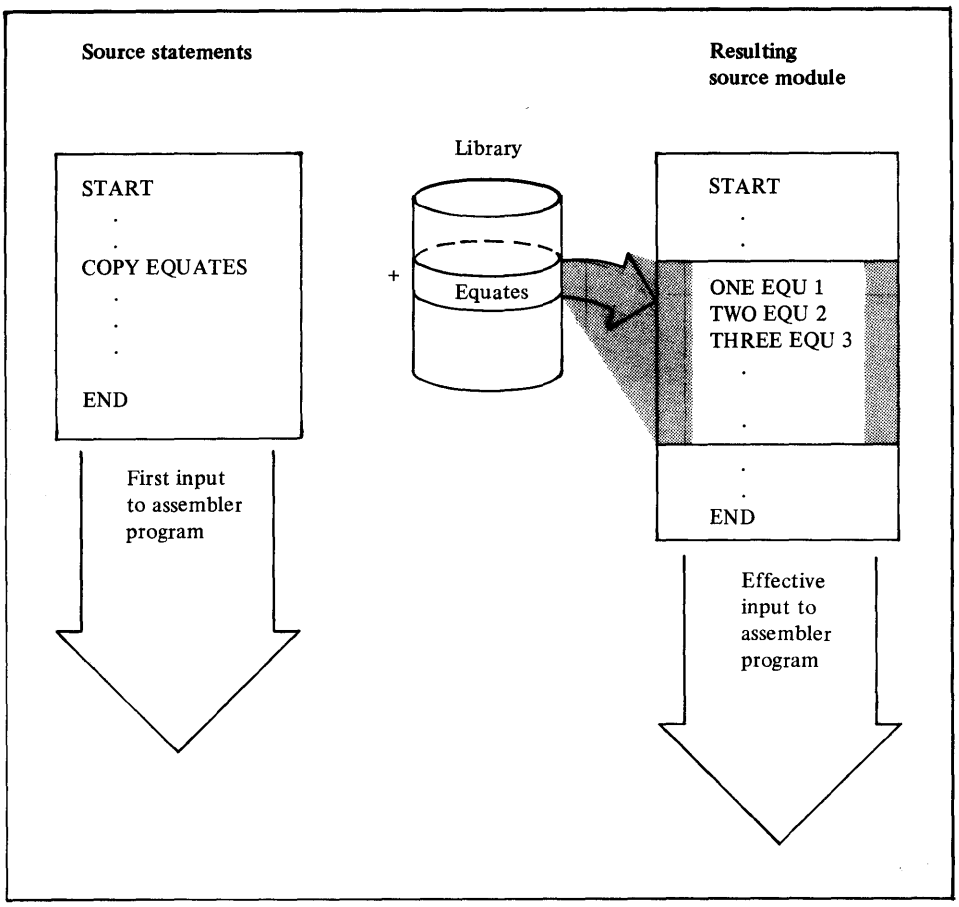
The End of a Source Module

The END instruction marks the end of a source module. Only one END instruction is allowed. The assembler does not process any instruction that follows the END instruction.

COPY—Copy Predefined Source Coding

The COPY instruction allows you to copy predefined source statements from a library and include them in your source module. You thereby avoid:

1. Writing repeatedly the same, often-used sequence of code
2. Keying or handling the source statements for that code.



Specifications

The format of the COPY instruction statement is shown next. The symbol in the operand field must identify:

- A member of a partitioned data set.

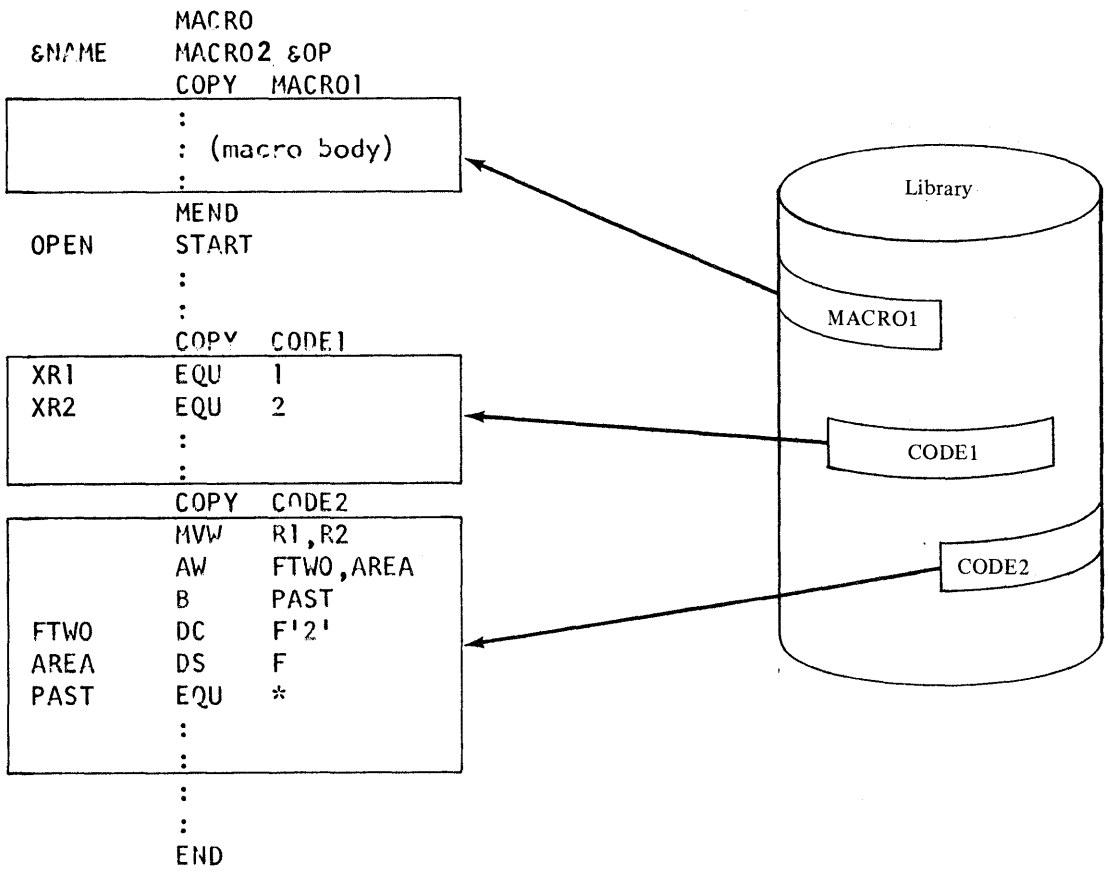
Name	*Operation*	*Operand*
Blank	COPY	A symbol

This member contains the coded source statements you want copied. The source coding that is copied into a source module:

- Immediately follows the COPY instruction
- Is inserted and processed according to the standard instruction statement coding format (described under "Coding Specifications" in Chapter 2), even if an ICTL instruction has been specified
- Must not contain COPY, END, ISEQ, or ICTL instructions.

Notes.

1. You can also copy statements into source or system macro definitions with the COPY instruction.
2. The rules that govern the occurrence of assembler language statements in a source module also govern the statements you copy into a source module.



END—End Assembly

The END instruction marks the end of a source module. It indicates to the assembler where to stop assembly processing.

You can also supply on the END instruction the address of the location in your program where execution must start. This location is quite often the address of the first executable instruction in the source module. In this case leave the operand blank. Later, if you wish, you can override this location with application build control statements.

Note. The entry address you specify in an application build control statement must be a CSECT name or a name defined in an assembler ENTRY instruction. The entry address in an END instruction can be any name defined in your source module.

The format of the END instruction statement is:

Name	*Operation*	*Operand*
blank	END	relocatable expression OR blank

If specified, the relocatable expression must meet one of the following conditions:

- It must be a relocatable expression representing an address in the source module delimited by the END instruction, or
- If it contains an external symbol, that symbol must be the only term in the expression, or the remaining terms in the expression must reduce to zero.

The following example indicates the use of the END instruction:

```
PROG    START
        :
ENTER   EQU   *
        :
        END   ENTER
```

General Information About Control Sections

A control section is the smallest subdivision of a program that can be relocated as a unit. The assembled control sections contain the object code for machine instructions and data.

Control Sections at Different Processing Times

Consider the concept of a control section at different processing times:

At Coding Time. You create a control section when you write the instructions it contains. In addition, you establish the addressability of each control section within the source module, and provide any symbolic linkages between control sections that are in different source modules.

At Assembly Time. The assembler translates the source statements in the control section into object code. Each source module is assembled into one object module. The entire object module and each of the control sections it contains is relocatable.

At Application Builder Time. Based on your control statements, the application builder combines the object code of one or more control sections into one load module. It also calculates the linkage addresses necessary for communication between two or more control sections from different object modules.

Types of Control Sections

An *executable control section* begins with the START or CSECT instructions and is assembled into object code. At execution time, an executable control section contains the binary data assembled from your coded instructions and constants.

Note. An executable control section is usually named. You can also initiate an executable control section as "private code" when you omit the START or CSECT instruction, or when you specify an unnamed START or CSECT instruction. (See "Unnamed Control Section" in this chapter.)

A *reference control section* begins with the DSECT, COM or GLOBL instruction and is not assembled into object code. You can use reference control sections to describe the contents of data areas to your executable control sections (DSECT) and also to reserve common storage (COM) or global storage (GLOBL).

Location Counter Setting

The assembler maintains a separate internal location counter for each control section so that they can be intermixed in your source module. The location counter for each control section is set to zero at the beginning of that control section. The location values assigned to the instructions and other data in a control section are relative to the beginning of that control section.

You can continue a control section that has been discontinued by another control section and thereby intersperse code sequences from different control sections. Note that the location values that appear in the listings for a control section, divided into segments, follow from the end of one segment to the beginning of the subsequent segment.

```
* SOURCE STATEMENTS                    LISTED LOCATION (HEXADECIMAL)
*
ONE       START 128                   ØØ8Ø
          :
          :                           Ø2Ø3
TWO       CSECT                       ØØØØ
          :
          :                           ØA43
THREE     CSECT                       ØØØØ
          :
          :                           ØCØØ
TWO       CSECT                       ØA44
          :
          :                           ØBØ1
          END
```

Length of Control Sections

The length of a control section is the sum of the extents of the first definition plus all continued control sections of the same name. The assembler maintains each control section on a byte address basis; that is, a control section may contain an odd number of bytes and a continued section need not resume at an even byte address. The application builder, however, will allocate an even number of bytes for all executable control sections.

First Control Section

The following specifications apply only to the first executable control section, and not to a reference control section:

Instruction that establish the first control section. Any instruction that affects the location counter or uses its current value establishes the beginning of the first executable control section. The instructions that establish the first control section are:

All machine instructions

```
CSECT
DC
DROP
DS
ALIGN
END
EQU
EQUR
ORG
PREF
PUSH
START
USING
```

These instructions are always considered a part of the control section in which they appear. The DSECT, COM and GLOBL instructions initiate reference control sections and do not establish the first executable control section.

What must come before the first control section. Source macro definitions, if specified, must appear before the first control section. (See Chapter 6.) The ICTL instruction, if used, must be the first statement in a source program.

What can optionally come before the first control section. The instructions or groups of instructions that can optionally be specified before the first control section are listed below. Any instruction you copy with a COPY instruction or generate with a macro instruction, before the first control section, must also belong to one of the following groups of instructions.

ICTL instruction
Macro definitions (must precede first control section)
COPY instruction
EJECT instruction
ENTRY instruction
EXTRN instruction
PRINT instruction
SPACE instruction
TITLE instruction
WXTRN instruction
Comments statements
Dummy control sections
Macro call (depends on expanded body of macro)
Common control sections
Global control sections

Notes.

1. These instructions belong to a source module, but are not considered as part of an executable control section.
2. TITLE, PRINT, SPACE, EJECT, ISEQ, and comment statements must follow your ICTL instruction, if specified. However, they can precede or appear between source macro definitions. All other instructions in your source module must follow any source macro definitions.
3. These instructions can all be coded within a control section.

Unnamed Control Section

Each source module can have only one unnamed control section. An unnamed control section is an executable control section you initiate in one of the following two ways:

- By coding a START or CSECT instruction without a name entry
- By coding any instruction (other than the START or CSECT instruction) that initiates the first executable control section

The unnamed control section is also referred to as *private code*. You should name all control sections so that you can refer to them symbolically:

- Within a source module
- In EXTRN, WXTRN, BX, and BALX instructions, for linkage between source modules

External Symbol Dictionary Entries

The assembler keeps a record of each control section and prints the following information about them in the external symbol dictionary:

- Symbolic name, if one is specified
- Type code
- ESD identification number
- Starting address
- Length in bytes

The following table lists:

- The assembler instructions that define control sections and dummy control sections, or identify entry and external symbols, and
- The type code that the assembler assigns to the control sections or dummy control sections, and to the entry and external symbols.

Instruction label	*Instruction*	*Type code entered into external symbol dictionary*
Optional	START	SD if label is present PC if label is omitted
Optional	CSECT	SD if label is present PC if label is omitted
Optional	Any instruction that initiates the unnamed control section	PC
Mandatory	DSECT	None
Blank	ENTRY	LD
Blank	EXTRN	ER
Blank	WXTRN	WX
Optional	DC (V type address constant)	ER
Optional	DC (W type address constant)	WX
Optional	DC (N type name constant)	RR
Optional	BX and BALX	ER
Optional	COM	CM
Optional	GLOBL	GL

Defining a Control Section

You must use the START, CSECT, COM, GLOBL, and DSECT instructions to indicate to the assembler:

- Where a control section begins, and
- What type of control section is being defined

START—Start Assembly

The START instruction can initiate only the first executable control section in your source module. You should use the START instruction for this purpose, because it allows you to:

- Determine exactly where the first control section begins, thereby avoiding the accidental initiation of the first control section by some other instruction.
- Give a symbolic name to the first control section, so you can distinguish it from the other control sections listed in the external symbol dictionary.
- Specify the initial setting of the location counter for the first or only control section.

The START instruction, when used, must be the first instruction of the first executable control section in your source module. You must not precede it with any instruction that affects the location counter and thereby causes the first control section to be initiated.

The format of the START instruction statement is:

Name	*Operation*	*Operand*
[label]	START	self-defining term OR blank

The symbol in the label of the START instruction identifies the first control section. You use the same symbol in the label of any CSECT instruction that resumes the first control section. This symbol represents the address of the first word in the control section. The assembler uses the value of the self-defining term you specify on the START instruction, to set the location counter initial value.

The value of the operand must be aligned to a word (divisible by 2). If you omit the operand entry, the assembler sets the location counter to zero. For example:

```
* SOURCE STATEMENTS                    LISTED LOCATION (HEXADECIMAL)
*
FIRST    START 256                     Ø1ØØ
         :
         :                             Ø11D
SECOND   CSECT                         ØØØØ
         :
         :                             Ø3ØØ
FIRST    CSECT                         Ø11E
         :
         :
         :                             Ø22B
         END
```

The source statements that follow the START instruction are assembled into the first control section. If a CSECT instruction indicates a continuation of the first control section, the source statements that follow this CSECT instruction are also assembled into the first control section.

Any instruction that defines a new or continued control section marks the end of the preceding control section or part of a control section. The END instruction marks the end of the last control section. For example:

```
FIRST    START Ø
         :
SECOND   CSECT
         :
DUMMY    DSECT
         :
FIRST    CSECT
         :
         END
```

CSECT—Start or Resume a Control Section

With the CSECT instruction, you initiate an executable control section or indicate the continuation of an executable control section.

You can use the CSECT instruction anywhere in a source module after your source macro definitions, if you have them. If you use CSECT to initiate the first executable control section, you must not precede it with any instruction that affects the location counter and thereby causes the first control section to be initiated.

The format of the CSECT instruction statement is:

Name	*Operation*	*Operand*
[label]	CSECT	blank

The symbol you specify in the label of the CSECT instruction identifies the control section. If you have several CSECT instructions within your source module with the same symbol in the name field, the first occurrence initiates the control section and the rest indicate the continuation of the same control section. If you initiate the first control section with a START instruction, use the symbol in its label to indicate a continuation of the first control section.

Note. Coding a CSECT instruction with a blank label either initiates or indicates the continuation of the unnamed control section. There can be only one unnamed control section in a source module.

The symbol in the label of the CSECT instruction represents the first word in the control section. The source statements following a CSECT instruction assemble into the object code of the control section identified by that CSECT instruction. The end of a control section, or part of a control section, is marked by:

- Any instruction that defines a new or continued control section, or
- The POP instruction
- The END instruction

DSECT—Start or Resume a Dummy Control Section

The DSECT instruction begins a dummy control section or indicates its continuation. A dummy control section is a reference control section that allows you to write a sequence of assembler language statements to describe the layout of data located elsewhere. The assembler produces no object code for statements in a dummy control section and it reserves no main storage. Rather, the dummy section provides a symbolic format for a data area in storage. The assembler assigns location values to the symbols you define in a dummy section, relative to the beginning of that dummy section.

Therefore, to use a dummy section you must:

- Reserve a storage area for the data in an executable control section of the same or another source module.
- Ensure that the data is in the area at execution time.
- Ensure that the locations of the symbols in the dummy section actually correspond to the locations of data in the area.
- Establish the addressability for the DSECT in combination with the storage area.

You can then refer to the data symbolically by using the symbols defined in the DSECT.

The symbol you specify in the label of the DSECT instruction identifies the dummy section. If you have several DSECT instructions within your source module with the same symbol in the name field, the first occurrence initiates the dummy section and the rest indicate the continuation of that dummy section.

The symbol in the label of the DSECT instruction represents the first location described in the dummy section. The location counter for a dummy section has an initial value of zero. However, the continuation of a dummy section begins at the next available location in that dummy section.

The format of the DSECT instruction statement is:

Name	*Operation*	*Operand*
label	DSECT	blank

The label of the DSECT instruction can be any ordinary symbol. The source statements that follow a DSECT instruction belong to the dummy section identified by that DSECT instruction. Assembler language statements that appear in a dummy control section do not assemble into object code.

When you establish addressability for a dummy section, the symbol in the label of the DSECT instruction, or any symbol defined in the dummy section can be specified in a USING instruction.

A symbol defined in a dummy section can be specified in an address constant only if the symbol is paired with another symbol from the same dummy section, and if the symbols have the opposite sign. For example:

```
* INSTRUCTIONS                    DESCRIPTION
*
MYPROG   START
         EXTRN DATA
         MVA   DATA,R7            ESTABLISH ADDRESSABILITY
         USING DUMMY,R7           NAME FIELD OF A DSECT STATEMENT
         :
ADCON    DC    A(FROM-TO)         PAIRED SYMBOLS DEFINED IN DSECT
         :
DUMMY    DSECT
         :
TO       DS    CL20               FIELDS NOT ASSEMBLED INTO
FROM     DS    CL60               OBJECT CODE
         END
```

COM—Start or Resume a Common Control Section

You use the COM instruction to initiate a common control section or to indicate its continuation. A common control section is a reference control section that allows you to reserve a storage area that can be shared by more than one source module within a task.

A common control section allows you to describe a common storage area in one or more source modules.

When the application builder combines separately assembled object modules into one program, the required storage space is reserved for the common control section. Thus, 2 or more modules can share common area which is defined with the same name in each module.

Only the storage area is provided; the assembler does not assemble the source statements that make up a common control section into object code. You provide the data for the common area at execution time.

The assembler assigns locations to the symbols you define in a common section relative to the beginning of that common section. This allows you to refer symbolically to data in the common area at execution time. If you code common sections in 2 or more source modules, you can communicate data symbolically between these modules through this common section.

Note. When you write a source module in a higher level language such as FORTRAN, you can also code a common control section. This allows you to communicate between assembler language modules and higher level language modules.

Specifications for COM

The COM instruction identifies the beginning or continuation of a common control section.

You can use the COM instruction anywhere in a source module after the ICTL instruction, and after your source macro definition if you have them.

The format of the COM instruction is:

Name	*Operation*	*Operand*
[label]	COM	Blank

The COM instruction either initiates or indicates the continuation of a common section. The location counter for a common section is always set to an initial value of zero. However, the continuation of a common section begins at the next available location in that section.

Note. If you specify a common section with the same name in 2 or more source modules, the application builder reserves the amount of storage for the common section equal to that required by your longest common section.

The source statements that follow a COM instruction belong to the common section identified by that COM instruction.

GLOBL—Start or Resume a Global Control Section

A GLOBL instruction is used to initiate a global control section or to indicate its continuation. A global control section is one of two types:

- An area that is shared across tasks linked to the shared task set (system global)
- An area that is shared by task sets which run within the same partition (partition global).

When the application builder combines object modules to form a task set, and a shared task set has been specified for resolving references, a check will be made for a match of the global section names against the shared task set. If a match is found in the shared task set, references to that global section will be resolved to it (system global). If no match is found, the global section becomes a part of partition global for the task set and the application builder will reserve storage for the section equal to that required by the longest global section with the same name.

Only the storage area is provided; the assembler does not create object code for the source statements that make up a global section. You provide the data for the global sections at execution time.

Specifications for GLOBL

The GLOBL instruction identifies the beginning or continuation of a global control section.

You can use the GLOBL instruction anywhere in a source module after the ICTL instruction, if any, and after any source macro definitions.

The format of the GLOBL instruction is:

Name	*Operation*	*Operand*
[label]	GLOBL	Blank

The location counter for the global section is always set to an initial value of zero. The continuation of a global section begins at the next available location in the section.

PUSH—Push Section

The PUSH statement saves information about the current control section in an internal assembler stack. The section may be restored later on a last-in, first-out basis by the use of a POP instruction. PUSH does not change the current section.

The format of the PUSH instruction is:

Name	*Operation*	*Operand*
blank	PUSH	SECTION

The name field of the PUSH instruction must be blank. SECTION is a required keyword. Use the PUSH instruction to:

- Save data about the current control section whether it has been initiated by a CSECT, DSECT, COM, GLOBL, START, or is private code. The section type, name, and ESDID are all saved by PUSH and restored by POP. Up to 16 sections may be concurrently on the stack.

PUSH can appear as often as required anywhere within a storage program. It will initate an unnamed control section, CSECT (private code), if it appears before the start of a control section.

POP—Pop Section

The POP instruction restores the control section to the section on the top of the internal assembler section stack.

The format of the POP instruction is:

Name	*Operation*	*Operand*
blank	POP	SECTION

The name field of the POP instruction must be blank. SECTION is a required keyword. Use the POP instruction to:

- Restore the section name, section type and ESDID of the last section PUSHed on the stack. It sets the location counter to the next available location in the section. POP can appear as often as required anywhere within a source

```
A       CSECT
        MVW   R1,R2
        XYZD  ,                    MACRO CALL
+       PUSH  SECTION              SAVE SECTION A
+XYZ    DSECT
+X1     DS    F
+X2     DS    F
+       POP   SECTION              RESUME SECTION A
        MVW   R2,R3
         .
         .
```

The macro XYZD saves the current section, CSECT A. It generates DSECT XYZ and then resumes CSECT A.

Symbolic Addressing Within Source Modules—Establishing Addressability

The assembler must be able to establish addressability for all machine instructions that reference a storage location. You can consider these instructions to belong to one of four groups:

- Instructions that reference a location using a register as a base, such as BALS, MVWS, and BXS.
- Instructions that reference a location by its effective address or by using a register as a base; this group includes all instructions with *addr4, addr5,* and *longaddr* formats, such as MVW, B, CW, and BAL. These instructions can reference any location within the range of the assembler's location counter (65535). You do not have to establish addressability for these instructions, even if the referenced location is not in the same control section as the instruction.

  Note. When you use the *addr* form of these instruction formats, the assembler generates RLD items which must be processed by the application builder. The base register-displacement format is self-relocating and does not need to be relocated by the application builder. RLD items are not generated for instructions or data in DSECT, COM, and GLOBL sections.
- Jump instructions that can only reference a location using the IAR as a base, such as J, JAL, and JCT. For this group, the referenced location must be relocatable and within the range −256 to 254 bytes of the byte following the jump instruction, and also within the same control section. You manually establish addressability for these instructions by ensuring that the referenced location is within IAR range. If the location is not within range, the assembler will flag the jump instruction.
- Instructions that refer to a location specified as the contents of a register. You do not have to establish addressability for these instructions.

You can establish addressability for the first two types of instructions in either of two ways:

- You can code an explicit address by coding the register-displacement form of the operand. This method requires that you develop absolute displacements from a location whose address you load into the register at execution time. Using EQU instructions permits you to develop symbolic displacements.
- You can let the assembler compute a displacement and index register combination that is suitable for referencing the required location.

Letting the assembler compute displacements has certain advantages over other methods of establishing addressability:

- All data constants and I/O buffers can be grouped together and separated from machine instruction logic at the end of your control section or they can be assembled as a separate control section.
- Fields can be symbolically referenced, thus improving code readability.

For the assembler to compute displacements from a register, you must, at coding time:

- Specify a base address from which the assembler can compute the displacements
- Assign a register to contain this base address
- Write the instruction that loads the register with the base address

At assembly time, the address operands you code are converted into their register-displacement form. Then they are assembled into the object code of the machine instructions in which you coded them.

At execution time, the base address must be loaded into the register and should remain there throughout execution of the code that depends on that address to locate the subject data locations.

The following example indicates the use of a base register to establish addressability.

```
* INSTRUCTIONS                          DESCRIPTION
*
MYPROG   START
         :
         MVA    DATA,R2                LOAD BASE ADDRESS INTO REGISTER
         USING  DATA,R2                SPECIFY BASE ADDRESS AND
*                                      ASSIGN REGISTER
         :
         MVWS   FIELD1,R3
         AW     FIELD2,R3
         MVWS   R3,FIELD3
         :
DATA     EQU    *
FIELD1   DS     F'0'
FIELD2   DS     F'2'
FIELD3   DS     F
         :
         END
```

USING—Use Base Address Register

The USING instruction allows you to specify a base address and assign a register. If you also load the register with the base address, you have established addressability for data located within, as a maximum, −32767 to +65535 bytes of the base address. To use the USING instruction correctly, you should:

- Know which locations in a control section are made addressable by the USING instruction.
- Know which instructions can use these addresses as operands.
- Know which instructions can use the specified register as a base register.

The range of a USING instruction (called the *USING range*) is a maximum of −32767 to +65535 bytes from the base address specified in the USING instruction. The range does not extend beyond the boundaries of the executable or reference control section in which the base address is defined. The assembler can convert only addresses that are within a USING range to their register-displacement form; those outside the USING range cannot be converted.

The USING range does not depend upon the position of the USING instruction in the source module; rather, it depends upon the location of the base address specified in the USING instruction.

Note. The USING range is the range of addresses within a control section that is associated with the register specified in the USING instruction.

The range of the USING instruction and the valid base registers vary according to individual instruction formats, as follows:

- All instructions with *addr4* operand formats:

Coded format	*Resulting register displacement format*	*USING range*
addr	(reg$^{1-3}$, waddr)	−32767 to +65535
addr*	(reg$^{1-3}$, disp)*	0 to 255

- All instructions with *addr5* operand formats:

Coded format	*Resulting register displacement format*	*USING range*
addr	(reg$^{1-7}$, waddr)	–32767 to +65535
addr*	(reg$^{1-7}$, disp)*	0 to 255

- All instructions with *longaddr* operand formats:

Coded format	*Resulting register displacement format*	*USING range*
addr	(reg$^{1-7}$, waddr)	–32767 to +65535
addr*	(reg$^{1-7}$, waddr)*	–32767 to +65535

- The MVWS instruction:

Coded format	*Resulting register displacement format*	*USING range*
addr	(reg$^{0-3}$, wdisp)	0 to 62
addr*	(reg$^{0-3}$, wdisp)*	0 to 62

- The BALS instruction:

Coded format	*Resulting register displacement format*	*USING range*
addr*	(reg, jdisp)*	–256 to +254

- The BXS instruction:

Coded format	*Resulting register displacement format*	*USING range*
addr	(reg$^{1-7}$, jdisp)	–256 to +254

Here is some sample code that illustrates the range of the USING instruction:

```
* INSTRUCTION                    DESCRIPTION
*
MYPROG   START
         :
         MVA   DATA,R2          LOAD BASE ADDRESS INTO REGISTER
         USING DATA,R2          SPECIFY BASE ADDRESS AND ASSIGN
*                               REGISTER
         :
         MVWS  FIELD1,R1        FIELD1 WITHIN USING RANGE SO
*                               ADDRESS CONVERTS PROPERLY
         :
         MVW   FIELD2,R6        CANNOT CONVERT ADDRESS EVEN THOUGH
*                               FIELD2 IS WITHIN 65535 BYTES OF DATA
*                               BECAUSE FIELD2 IS NOT IN THE SAME
*                               CONTROL SECTION
         :
DATA     CSECT                  USING RANGE STARTS HERE AND IS 65535
*                               BYTES OR LESS, DEPENDING ON THE
*                               BOUNDARIES OF THE CONTROL SECTION
*                               AND THE INSTRUCTIONS CODED
FIELD1   DS    F
         :
SECOND   CSECT
FIELD2   DS    F
         :
         END
```

The domain of a USING instruction (called the *USING domain*) begins where the USING instruction appears in a source module and continues to the end of the source module. (Exceptions are discussed later in this chapter in "Notes About the Using Domain.") The assembler converts addresses in instructions into register-displacement form only when:

- The instructions appear in the domain of a USING instruction, and
- The addresses referred to are within the range of the same USING instruction.

The USING domain depends on the position of the USING instruction in the source module after macro expansion, if any, occurred.

```
* INSTRUCTIONS                          DESCRIPTION
*
MYPROG   START
         :
         MVA   DATA,R2
         MVW   FIELD,R6                 CANNOT CONVERT ADDRESS
         :
         USING DATA,R2                  USING DOMAIN STARTS HERE
         MVW   FIELD,R5                 CAN CONVERT ADDRESS
         :
DATA     CSECT
FIELD    DC    X'10'
         :
         END                            USING DOMAIN ENDS HERE
```

You should specify your USING instructions so that:

- As many data items as possible are grouped within a USING range, and

- All the instructions that refer to these data locations are within the corresponding USING domain.

You should therefore place USING instructions at the beginning of your coded instruction sequences and specify a base address in each USING instruction for each USING range you require. You can use the same register in multiple USING instructions so long as you load the register each time the required address changes.

For Executable Control Sections. The next example shows a way to establish addressability for an executable control section. The USING domain starts with the USING instruction and continues to the END instruction; the USING range (maximum) is from 32767 bytes before the EQU instruction to 65535 bytes after the EQU instruction.

```
* INSTRUCTIONS                          DESCRIPTION
*
          :
          MVA   DATA,R2                 LOAD BASE ADDRESS INTO REGISTER
          USING DATA,R2                 SPECIFY BASE ADDRESS AND ASSIGN
*                                       REGISTER
          :
          :                             MACHINE INSTRUCTIONS HERE
          :
DATA      EQU   *
          :                             DATA ITEMS HERE
          END
```

For Reference Control Sections. The next example shows how to establish addressability for a dummy section (a reference control section defined by a DSECT instruction). The address you load into the register at execution time must be the base address specified in the USING instruction. Note that the assembler assumes you are referring to the symbolic addresses in the dummy section, and it computes displacements accordingly. However, at execution time, the assembled addresses refer to the location of real data in the storage area. The USING range in the next example is the reference control section—from the DSECT instruction to the END instruction. The USING domain is from the USING instruction to the END instruction.

```
* INSTRUCTIONS                          DESCRIPTION
*
          :
          MVW   ADCON,R1                LOAD BASE ADDRESS INTO REGISTER
          USING DUMMY,R1                SPECIFY BASE ADDRESS AND ASSIGN
*                                       REGISTER
          :
ADCON     DC    V(EXTERNAL)
          :
          MVW   FIELD,R6
          :
DUMMY     DSECT                         USING RANGE STARTS HERE
FIELD     DS    F
          :
          END                           USING RANGE ENDS HERE
```

USING Instruction Format

The format of the USING instruction is:

Name	*Operation*	*Operand*
blank	USING	addr, reg

The name field of the USING instruction must be blank. The address specifies a base address, which must be a relocatable expression. The value of the expression must be in the range 0–65535. The register can be specified by an absolute register expression whose value is in the range 0–7. The assembler assumes that the register contains the base address at execution time (the USING instruction does not load the address into the register).

Coding Note. If you use the MVA instruction to load the base register, code it before the USING instruction. That prevents the following error:

```
          USING DATA,R1
          MVA   DATA,R1
          :
DATA      EQU   *
```

In this example, the MVA is in the domain of the USING, so the assembler computes a displacement of 0 for DATA, then generates the equivalent of:

```
MVA (R1,0),R1
```

Since R1 does not already contain the address of DATA, unpredictable results will occur at execution time whenever R1 is used as a base register. Code this instead:

```
          MVA   DATA,R1
          USING DATA,R1
          :
DATA      EQU   *
```

Notes About the USING Domain. The domain of a USING instruction continues until the end of a source module except when:

- A subsequent DROP instruction specifies the same register assigned by the USING instruction.
- A subsequent USING instruction specifies the same register assigned by the preceding USING instruction.

In the following example, instructions cannot be converted to register-displacement form between the DROP instruction and the second USING instruction.

```
* INSTRUCTIONS                    DESCRIPTION
*
          :
          USING DATA,R1           FIRST USING DOMAIN STARTS HERE
          :
DATA      EQU   *
          :
          DROP  R1                FIRST USING DOMAIN ENDS HERE
          :
DATA2     EQU   *
          :
          USING DATA,R2           SECOND USING DOMAIN STARTS HERE
          :
          USING DATA2,R2          SECOND USING DOMAIN ENDS HERE, AND
*                                 THIRD USING DOMAIN STARTS HERE
          :
          END                     THIRD USING DOMAIN ENDS HERE
```

Notes About the USING Range. Two USING ranges coincide when the same base address is specified in two different USING instructions, even though the registers are different. When two USING ranges coincide, the assembler uses the lower numbered register for assembling the addresses within the common USING range. (The first USING domain terminates at the second USING instruction.)

```
* INSTRUCTIONS                    DESCRIPTION
*
COINCIDE START
          :
          USING DATA,R2
          :                       INSTRUCTIONS HERE USE R2 AS
          :                       A BASE REGISTER
          USING DATA,R1
          :                       INSTRUCTIONS HERE USE R1 AS
          :                       A BASE REGISTER
DATA      EQU   *
          :                       INSTRUCTIONS HERE USE R1 AS
          :                       A BASE REGISTER
          END
```

Two USING ranges overlap when the range of one USING instruction is within the range of another USING instruction. When two ranges overlap, the assembler computes displacements from the base address using the lower-numbered register when it assembles the addresses within the range overlap. This applies only to instructions that appear after the second USING instruction.

DROP—Drop Base Register

The DROP instruction terminates the USING domain for one or more registers. Use the DROP instruction to:

- Free registers for other purposes.
- Ensure that the assembler uses the base register desired in a particular coding situation (as when two USING ranges overlap or coincide, as described in "Notes About the Using Range.")

The format of the DROP instruction is:

Name	*Operation*	*Operand*
blank	DROP	1–8 absolute register expressions, separated by commas

The name field of the DROP instruction must be blank. Up to 8 register expressions can be specified on one DROP instruction; the expressions must be absolute with a value in the range 0–7.

After a DROP instruction, the assembler no longer uses the dropped register as a base register. A register made unavailable as a base register by a DROP instruction can be reassigned as a base register with a subsequent USING instruction. For example:

```
* INSTRUCTIONS                          DESCRIPTION
*
          :
          USING DATA,R2
          :                             R2 AVAILABLE FOR USE AS A
          :                             BASE REGISTER HERE
          DROP  R2
          :                             R2 UNAVAILABLE FOR USE
          :                             AS A BASE REGISTER HERE
          USING DATA,R2
          :                             R2 AVAILABLE FOR USE AS A
          :                             BASE REGISTER HERE
DATA      EQU   *
          :
          END
```

You need *not* use a DROP instruction:

- If you reassign a register in a new USING instruction (however, you must load the new base address into the register).
- At the end of a source module.

```
* INSTRUCTIONS                          DESCRIPTION
*
          :
          MVA   DATA,R1                 LOAD BASE ADDRESS INTO REGISTER
          USING DATA,R1                 SPECIFY BASE ADDRESS AND
*                                       ASSIGN REGISTER
          :
          MVA   DATA2,R1                LOAD NEW BASE ADDRESS INTO REGISTER
          USING DATA2,R1                SPECIFY NEW BASE ADDRESS AND
*                                       ASSIGN REGISTER
          :
DATA      EQU   *
          :
DATA2     EQU   *
          :
          END
```

Symbolic Addressing Between Source Modules—Symbolic Linkage

This section describes symbolic linkage; that is, using symbols to communicate between different source modules that are separately assembled and then linked together by the application builder.

To establish symbolic linkage with an external source module:

1. You must identify the symbols that are not defined in your source module. These symbols are called *external* symbols, because they are defined in another (external) source module. You can identify external symbols:
   - Explicitly with the EXTRN or WXTRN instruction
   - Implicitly with the V- or W-type address constants
   - With the BALX and BX machine instructions
2. You must provide the A-, V-, or W-type address constants so the assembler can reserve storage for the addresses of the external symbols. When you use a BALX or BX instruction you do not provide an address constant; the address area is part of the instruction.
3. To resolve linkages, you must identify the symbols in the external source modules where you have them defined. These symbols are called *entry* symbols because they provide points of entry to a source module. You identify entry symbols with the ENTRY, CSECT, or START instruction.

The assembler places information about entry and external symbols in the external symbol dictionary. The application builder uses this information in conjunction with the relocation dictionary to resolve the linkages.

The following example illustrates symbolic linkage between three source modules.

```
* INSTRUCTIONS                          DESCRIPTION
*
REFERX   START ,                        START OF FIRST SOURCE MODULE
         EXTRN ONE,TWO
         WXTRN THREE
         :
         BALX  FOUR,R7
         :
ADCONS   EQU   *
         DC    A(ONE,TWO,THREE)
         DC    V(FIVE)
         DC    W(SIX)
         :
         END   ONE                      END OF FIRST SOURCE MODULE

DEFINE   START ,                        START OF SECOND SOURCE MODULE
         ENTRY ONE,TWO,THREE,FOUR
         :
ONE      EQU   *
         :
TWO      EQU   *
         :
THREE    EQU   *
         :
FOUR     EQU   *
         :
         END ,                          END OF SECOND SOURCE MODULE

DEFINE2  START ,                        START OF THIRD SOURCE MODULE
         ENTRY FIVE,SIX
         :
FIVE     EQU   *
         :
SIX      EQU   *
         :
         END ,                          END OF THIRD SOURCE MODULE
```

To Refer to External Data

You should use the EXTRN instruction to identify the external symbol that represents data in an external source module, if you wish to refer to this data symbolically.

For example, you can identify the address of a data area in an external source module as an external symbol and load the address constant for this symbol into a register. Then you may use this register when establishing the addressability of a dummy section (DSECT) that defines this external data area. You can now refer symbolically to data in the external area. You must also identify, in the source module that contains the data area, the same relative address of the data as an entry symbol.

In the following example, FIELD3 is assembled as part of the DEFINE source module (second source module); a dummy section in the REFERX source module (first source module) is used to refer to FIELD3; and, after link-editing the two source modules together, both source modules can access FIELD3.

```
* INSTRUCTIONS                          DESCRIPTION
*
REFERX   START,                         FIRST SOURCE MODULE STARTS HERE
         EXTRN DATA
         MVW   ADCON,R2
         USING DUMMY,R2
         :
         MVW   R3,FIELD3                FIELD3 REFERRED TO HERE
         :
ADCON    DC    A(DATA)
         :
DUMMY    DSECT                          DUMMY SECTION STARTS HERE
         DS    2F
FIELD3   DS    F                        FIELD3 DEFINED HERE
         :
         END,                           FIRST SOURCE MODULE ENDS HERE

DEFINE   START ,                        SECOND SOURCE MODULE STARTS HERE
         ENTRY DATA
         :
DATA     EQU   *
FIELD1   DC    F'1'
FIELD2   DC    X'FFØØ'
FIELD3   DC    F'Ø'                     FIELD3 ASSEMBLED HERE
         :
         END ,                          SECOND SOURCE MODULE ENDS HERE
```

To Branch to an External Address

You can use the BALX or BX machine instruction to branch to a location in an external source module. Code the external symbol as an operand in these instruction types.

You can also use the V-type address constant to identify the external symbol. For example, you can branch to an external address by branching indirectly with the V-type address constant. For the specifications of the V-type address constant, see "Defining Data" in this chapter.

If the external symbol is the label of a START or CSECT instruction in the other source module, and thus names an executable control section, it is automatically identified as an entry symbol. If the symbol represents an address in the middle of a control section, you must, however, identify it as an entry symbol in the external source module. For example:

```
* INSTRUCTIONS                    DESCRIPTION
*
REFERX  START ,                   START OF FIRST SOURCE MODULE
        :
        BX    SUBRTN              BRANCH TO EXECUTE SUBROUTINE
        :
        B     ADCON*              BRANCH TO EXECUTE MOD2
ADCON   DC    V(MOD2)             ADDRESS OF MOD2
        :
        END ,                     END OF FIRST SOURCE MODULE

MOD2    START ,                   START OF SECOND SOURCE MODULE
        ENTRY SUBRTN
        :
SUBRTN  EQU   *
        :
        END ,                     END OF SECOND SOURCE MODULE
```

You can also use a combination of an EXTRN instruction to identify, and an A-type address constant to contain the external branch address. However, the external branch instruction, or the V- or W-type address constants, is more convenient because you do not have to code an EXTRN instruction. With external branch instructions, you also do not code an address constant.

The assembler does not consider the symbol in an external branch instruction, a V-type or W-type constant, as defined in the source module. Therefore, you can use the same symbol as the name for most statements in the same source module, even the DC statement defining the V-type or W-type constant.

ENTRY—Identify Entry Point Symbol

ENTRY identifies symbols defined in the source module containing the ENTRY instruction so that you can refer to them in another source module. These symbols define locations that are called entry points.

The format of the ENTRY instruction is:

Name	*Operation*	*Operand*
blank	ENTRY	one or more relocatable symbols (entry symbols), separated by commas

The label of an ENTRY instruction must be blank.

The following rules apply to entry symbols:

- They must be valid symbols.
- You must define them in an executable control section within the current assembly. A symbol can appear on multiple ENTRY statements within an assembly.

A symbol used as the label of a START or CSECT instruction is also automatically considered as an entry point and does not have to be identified by an ENTRY instruction. Thus, in the following example, the two entry points are FIRST and SUBRTN.

```
FIRST    START
         ENTRY SUBTRN
         :
SUBTRN   EQU   *
         :
         END
```

The assembler lists each entry symbol in the external symbol dictionary, along with other entries for external symbols.

EXTRN—Identify External Symbol

EXTRN identifies symbols referred to in the source module containing the EXTRN instruction but defined in another source module. These symbols are called external symbols.

The format of the EXTRN instruction is:

Name	*Operation*	*Operand*
blank	EXTRN	one or more relocatable symbols (external symbols), separated by commas

The label of an EXTRN instruction must be blank.

The following rules apply to the external symbols:

- They must be valid symbols.
- You must not use them as the name entry of any source statement in the same source module.
- You must use them alone and not pair them in an expression, except within A-type address constants.

The assembler lists each external symbol in the external symbol dictionary, along with entries for entry symbols.

The following example indicates the relationship of ENTRY and EXTRN statements. Note that FOURTH need not be specified in an EXTRN statement since it is a V-type address constant in FIRST, and that SECOND need not be specified on an ENTRY statement since it is the label on the START statement.

```
FIRST    START
         EXTRN SECOND,THIRD
         :
EXTAD1   DC    A(SECOND)
EXTAD2   DC    A(THIRD)
EXTAD3   DC    V(FOURTH)
         :
         END

SECOND   START
         ENTRY THIRD,FOURTH
         :
THIRD    EQU   *
         :
FOURTH   EQU   *
         :
         END
```

WXTRN—Identify Weak External Symbol

WXTRN identifies symbols in the source module containing the WXTRN instruction but defined in another source module. The WXTRN instruction differs from the EXTRN instruction as follows:

The EXTRN instruction causes the application builder to make an automatic search of libraries to find the module that contains the external symbols. If a module is found, linkage addresses are resolved when the module is linked to yours.

The WXTRN instruction suppresses this automatic search of libraries. The application builder will resolve the linkage addresses only if the external symbols in the WXTRN operand field are defined:

- In a module that is linked to your object module because of application builder control statements
- In a module brought in from a library due to the presence of an EXTRN instruction in another module linked to yours.

The format of the WXTRN instruction statement is:

Name	*Operation*	*Operand*
blank	WXTRN	one or more relocatable symbols (weak external symbols), separated by commas

The label of a WXTRN instruction must be blank.

To the assembler, the external symbols identified by a WXTRN instruction have the same properties as the external symbols identified by the EXTRN instruction. However, the type code assigned to these external symbols in the external symbol dictionary is different. Also, the automatic library call mechanism (AUTOCALL) of the application builder is not activated for "weak" external references.

If you specify a symbol in a V-type address constant and also in a WXTRN instruction in the same source module, the symbol is processed as a weak external reference. If you specify an external symbol by both an EXTRN and WXTRN instruction in the same source module, the first declaration takes precedence, and subsequent declarations are flagged with error messages. You may use the same symbol in multiple EXTRN and external branch instructions. You may also duplicate a symbol in WXTRN instructions.

```
* INSTRUCTIONS                    DESCRIPTION
*
FIRST    START                    ESD TYPE FOR FIRST IS SD
         EXTRN OUT,A              ESD TYPE FOR OUT IS ER
*                                 ESD TYPE FOR A IS ER
         EXTRN B,OUT              ESD TYPE FOR B IS ER
         WXTRN WOUT               ESD TYPE FOR WOUT IS WX
         WXTRN A                  ***ERROR***
VCON     DC    V(WOUT)            ***ERROR***
WCON     DC    W(WOUT)
         :
         END
```

ORG—Set Location Counter

The ORG instruction alters the setting of the location counter and thus controls the structure of the current control section. This allows you to redefine parts of a control section.

The ORG instruction can cause the location counter to point to any part of a control section, where you can assemble desired data. It can also cause the location counter to point back to the next available location so that your program can continue to be assembled in a sequential fashion.

The format of the ORG instruction is:

Name	*Operation*	*Operand*
blank	ORG	relocatable expression OR blank

The label of an ORG instruction must be blank. The symbols in the relocatable expression must be previously defined in the source module. If the expression contains an unpaired relocatable term, you must define that term in the same control section in which the ORG statement appears. The location counter is set to the value of the relocatable expression. If the expression is omitted, the location counter is set to the next available location for the current control section. The following sample code illustrates the setting of the location counter with the ORG instruction:

```
* INSTRUCTIONS                          LISTED LOCATION (HEXADECIMAL)
*
MYPROG    START                         ØØØØ
          :
          BAL     INITIAL,R7
          :
          B       CONTINUE
BUFFER    DS      CL4Ø                  Ø1ØA
          DS      F                     Ø132
          ORG     BUFFER                Ø1ØA
INITIAL   EQU     *                     Ø1ØA
          :
          BXS     (R7)                  Ø12C
          ORG                           Ø134
CONTINUE  EQU     *                     Ø134
          :
          END
```

You must not specify an expression on an ORG instruction for a location that precedes the beginning of the control section in which the ORG appears. In the next example, the ORG instruction is invalid if it appears less than 100 bytes from the beginning of its control section. This is because the resulting expression would be negative and therefore invalid.

```
FIRST     START
          :
          ORG     *-1ØØ
          :
          END
```

Note. Using the ORG instruction to insert data at the same location as earlier data does not always work. In the next example, it appears as if the character constant overlays the address constant. However, after the application builder places the character constant into the same location as the address constant, it adds the relocation factor required for the address constant to the value of the constant. This sum is the object code that resides in the word ADDR.

```
ADDR    DC    A(LOC)
        ORG   *-2
CHAR    DC    C'BE'
```

You will experience unpredictable results when you code an ORG statement to insert data in any relocatable machine instruction.

ALIGN—Align Location Counter

The ALIGN instruction ensures the setting of the location counter to an odd byte, or word address during program assembly. This instruction is used primarily for data alignment.

The format of the ALIGN instruction is:

Name	*Operation*	*Operand*
blank	ALIGN	{ WORD / ODD }

The name field of the ALIGN instruction must be blank. WORD specifies that the location counter is to be reset if necessary to the next higher address which is evenly divisible by 2. ODD specifies that the location counter is to be reset if necessary to the next higher address which is not divisible by 2 (an odd byte boundary).

The ALIGN instruction can appear as often as required in a source program, but must not precede the start of the program control section.

Note. When the location counter is set to the required boundary address before ALIGN instruction processing, the ALIGN instruction is ignored. When the ALIGN instruction causes the location counter to be advanced, binary zeros are placed in the vacated byte positions.

Determining Statement Format and Sequence

You can change the standard coding conventions for the assembler language statements or check the sequence of source statements with the following instructions.

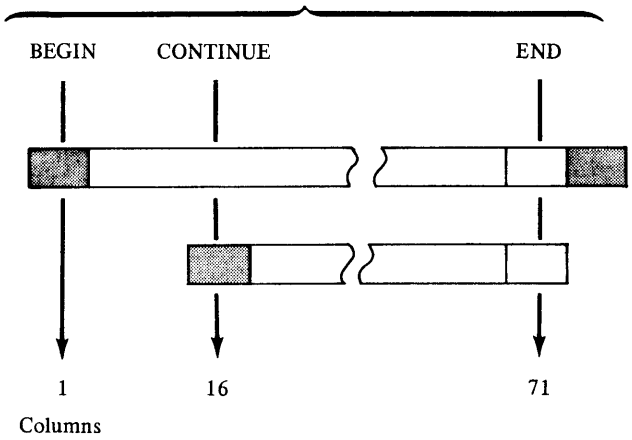
ICTL—Input Format Control

The ICTL instruction allows you to change the begin, end, and continue columns to establish a different coding format for your assembler language source statements.

For example, with the ICTL instruction, you can increase the number of columns for the identification or sequence checking of your source statements. By changing the begin column, you can even create a field before the begin column to contain identification or sequence numbers.

You can code the ICTL instruction only once, at the very beginning of your source module. If you do not code it, the assembler recognizes the standard values for the begin, end, and continue columns.

Standard values for columns



If you code the ICTL instruction, it must be the first statement in your source module. The format of the ICTL instruction statement is:

Name	*Operation*	*Operand*
Blank	ICTL	One to three decimal self-defining values of the form b,e,c

Operands	**Specifies**	**Allowable range**
b	Begin column	1 through 40
e	End column	41 through 80
c	Continue column	2 through 40

Rules for interaction of b, e and c	
The position of the End column must not be less than the position of the Begin column + 5, but must be greater than the position of the Continue column	$e \geq b + 5$ $e > c$
The position of the Continue column must be greater than that of the Begin column	$c > b$

The operand entry must be one to 3 decimal self-defining terms. There are only 3 possible ways you can specify the operand entry:

- begin
- begin,end
- begin,end,continue.

The operand *begin* must always be specified. The operand *end*, when not specified, is assumed to be 71. If the operand *continue* is not specified, or if *end* is specified as 80, the assembler assumes that continuation lines are not allowed. The values specified for the 3 operands depend on each other.

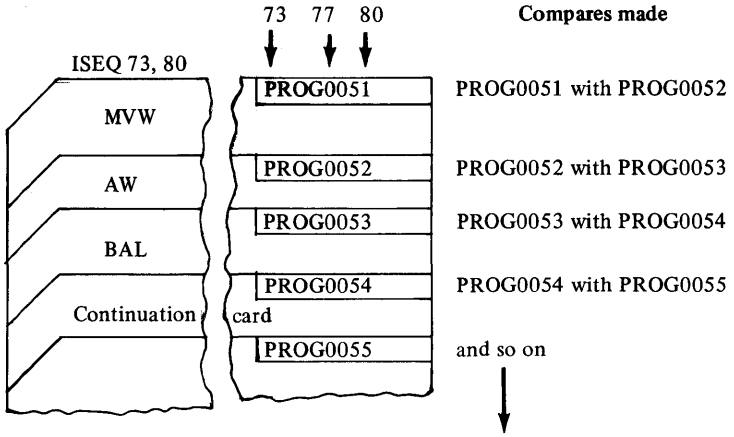
Note. The ICTL instruction does not affect the format of statements brought in by a COPY instruction or generated from a library macro definition. The assembler processes those statements according to the standard begin, end, and continue columns.

ISEQ—Input Sequence Checking

You use the ISEQ instruction to cause the assembler to sequence check the statements in your source module. In the ISEQ instruction you specify the columns you want the assembler to check for sequence numbers.

The assembler begins sequence checking with the first statement line following the ISEQ instruction. The assembler also checks continuation lines.

Sequence numbers on adjacent statements or lines are compared according to the internal EBCDIC collating sequence. When the sequence number on one line is not greater than the sequence number on the preceding line, a sequence error is flagged, and a warning message is issued, but the assembly is not terminated.



The ISEQ instruction initiates or terminates the checking of the sequence of statements in a source module. The format of the ISEQ instruction is:

Name	*Operation*	*Operand*
Blank	ISEQ	Two decimal self-defining values of the form l,r or blank

Operand	Specifies	Rules for interaction
l	leftmost column of field to be checked	l must not be greater than r
	l and r not allowed between begin and end columns	
r	rightmost column of field to be checked	r must not be less than l

When the operand field specifies 2 self-defining terms, the ISEQ instruction initiates sequence checking, beginning with the statement following the ISEQ instruction.

When the operand field is blank, the ISEQ instruction terminates the sequence checking operation. This terminating ISEQ instruction is also sequence checked.

Note. The assembler checks only those statements that are in your source module. This includes COPY instructions.

However, the assembler does not check:

- Statements inserted by a COPY instruction
- Statements generated from model statements inside macro definitions (Statement generation is discussed in detail in Chapter 6)
- Statements in library macro definitions.

Listing Format and Output

The PRINT, TITLE, EJECT, and SPACE instructions request the assembler to produce listings and identify records in the object module according to your special needs. They allow you to determine printing and page formatting options other than the ones the assembler program assumes by default. Among other things, you can introduce your own page headings, control line spacing, and suppress unwanted detail.

PRINT—Print Optional Data

PRINT controls the amount of detail you want printed in the listing of your program. The three options that you can set are given in the following table. They are listed in hierarchic order; if OFF is specified, GEN and DATA do not apply. If NOGEN is specified, DATA does not apply to the constants in generated statements. The standard options inherent in the assembler program are ON, GEN, and NODATA.

Hierarchy	*PRINT options*	*Description*
1	ON	A listing is printed
1	OFF	No listing is printed
2	GEN	All statements generated by the processing of a macro instruction are printed
2	NOGEN	Statements generated by the processing of a macro instruction are not printed. (*Note.* the MNOTE instruction always causes a message to be printed)
3	DATA	Constants are printed in full in the listing
3	NODATA	Only the leftmost 8 bytes of constants are printed in the listing

The format of the PRINT instruction is:

Name	*Operation*	*Operand*
blank	PRINT	ON GEN DATA OFF NOGEN NODATA

The label of the PRINT instruction must be blank. At least one of the print options must be specified, and at most one of the options from each group. If more than one option is specified, they must be separated by commas.

The options can be specified in any order. The PRINT instruction can be specified any number of times in a source module. At assembly time, all options are in force until the assembler encounters a new and opposite option in a PRINT instruction.

Note. The option specified in a PRINT instruction takes effect after the PRINT instruction. If PRINT OFF is specified, the PRINT instruction itself is printed, but not the statements that follow it.

TITLE—Identify Assembly Output

TITLE provides headings for each page of the assembly listing.

The format of the TITLE instruction is:

Name	*Operation*	*Operand*
id characters	TITLE	character string up to 100 characters, enclosed in apostrophes

The label field of the *first* TITLE instruction in a program can contain identification characters. Up to four identification characters are printed in the top left-hand corner of every page of the listing. Specifying a valid ordinary symbol in this field does not constitute a definition of that symbol for the source module.

The character string on the TITLE instruction is printed as a heading at the top of each page of the assembly listing. The heading is printed beginning on the page in the listing following the page on which the TITLE instruction is specified. A new heading is printed when a subsequent TITLE instruction appears in the source module.

Any printable character specified will appear in the heading, including blanks. However, the following rules apply to apostrophes:

- A single apostrophe followed by one or more blanks simply terminates the heading prematurely. If a nonblank character follows a single apostrophe, the assembler issues an error message and does not print a heading.
- Double ampersands or apostrophes print as a single ampersand or apostrophe in the heading.

Only the characters printed in the heading count toward the maximum of 100 characters allowed.

Note. The TITLE statement itself is not printed in an assembly listing.

EJECT—Start New Page

EJECT stops the printing of the assembly listing on the current page and continues the printing on the next page.

The format of the EJECT instruction is:

Name	*Operation*	*Operand*
blank	EJECT	blank

The label on an EJECT instruction must be blank. The EJECT instruction causes the next line of the assembly listing to be printed at the top of a new page. If the line before the EJECT instruction appears at the bottom of a page, the EJECT instruction has no effect.

Note. The EJECT instruction is not printed in the listing.

SPACE—Space Listing

SPACE inserts one or more blank lines in the listing of a source module. This allows you to separate sections of code on the listing page.

The format of the SPACE instruction is:

Name	*Operation*	*Operand*
blank	SPACE	decimal value from 1 to 255 OR blank

The label on a SPACE instruction must be blank.

The decimal value on the SPACE instruction specifies the number of lines to be left blank. A blank causes one blank line to be inserted. If the value specified is greater than the number of lines remaining on the listing page, the instruction has the same effect as an EJECT statement.

Note. The SPACE instruction is not printed in the listing.

Chapter 6. Macro Language

Section Contents

Macros are used mainly to insert defined groups of assembler language statements into a source program. The defined group of statements is a *macro definition;* the statements are either stored in a library or placed at the beginning of the source module. This chapter explains how to prepare macro definitions as the first portion of your source module.

The statements contained in the macro definition are *called* by macro instructions in your source program; the macro instruction is coded at the point you would otherwise include the statements contained in the macro definition. The calling macro instruction can specify parameters which change the statements contained in the macro definition. The assembler inserts the macro definition statements, as modified by the parameters on the calling macro instruction, immediately after the calling macro instruction. The process of inserting the text of the macro definition is called *macro generation* or *macro expansion* The expansion occurs for each macro instruction that calls the macro definition.

The assembler processes the source module in two phases:

- First, *preassembly*; during this phase the assembler expands macro calls by inserting text from macro definitions inline after the calling macro instructions. The statements contained in the macro definition can be modified during preassembly by assembler action as follows:
  - Processing *symbolic parameters* specified on the calling macro instruction to modify the statements in a macro definition which contain the same symbolic parameters.
  - Processing *conditional assembly instructions* contained within a macro definition; these instructions declare and assign values to *SET symbols* (symbols used to write source statements that can be modified during expansion) and allow for branching and loop control within a macro expansion. With conditional assembly language instructions, you can select and reorder the statements generated each time a macro is expanded.
  - Processing MNOTE instructions, which produce error messages that you provide.
  - Processing *system variable symbols;* these symbols can be used in macro definitions to cause the assembler to perform specific actions (for example, to count the number of symbolic parameters on a calling macro instruction so that the number can be used to determine further expansion of the macro).
- Then, *assembly*; during this phase the assembler processes the source module (known as *open code*) and the statements generated during macro expansion at preassembly time, to produce the object module.

By using the macro language you reduce programming effort, because:

- You write and test the code for a macro definition only once. You and other programmers can then use the same code as often as you like by calling the definition; this means that you do not have to reconstruct the coding logic each time you use the code.
- You need write only one macro instruction to call for the generation of many assembler language statements from the macro definition. Appendix A 'Structured Programming Macros' gives an example of using structured macros.

When you are designing and writing large assembler language programs, the above features allow you to:

- Change the code in one place when updating or making corrections, that is, in the macro definition. Each call gets the latest version automatically, thus providing standard coding conventions and interfaces.
- Describe the functions of a complete macro definition rather than the function of each individual statement it contains, thus providing more comprehensible documentation for your source module.

Overview of Creating Macros

You can create a macro definition by enclosing any sequence of assembler language statements between MACRO and MEND statements, and by writing a prototype statement in which you give your definition a name. This name is then the operation code that you must use in the macro call.

When you code a macro call in your source module, you tell the assembler to process a particular macro definition. The assembler generates assembler language statements from this macro definition for each occurrence of the macro call; if you code four calls to the macro MYMACRO, four sets of assembler language statements are generated. The statements generated can be:

- Copied directly from the definition
- Modified by parameter values before generation
- Manipulated by internal macro processing to change the sequence in which they are generated
- Selectively chosen or discarded in groups

You can define your own macro definitions in which any combination of these processes can occur. Some macro definitions do not generate assembler language statements, but perform only internal processing.

The MACRO and MEND instructions establish the boundaries of a macro definition. The prototype statement establishes the name of the macro and declares its parameters. In the operand field of the calling macro instruction, you can assign values to the parameters declared for the called macro definition. The body of a macro definition contains the statements that will be generated when you call the macro. These statements are called *model statements*; they are usually interspersed with conditional assembly statements or other processing statements.

You can include a macro definition at the beginning of a source module. This type of definition is called an *inline macro definition.* You can also insert a macro definition in a *library*. This type of definition is called a *library macro definition.*

The following example indicates the general format of a macro definition within a source module:

```
        MACRO                       MACRO HEADER
        MACID &PARAM1,&PARAM2       PROTOTYPE STATEMENT
        :
        :                           (Body of macro definition)
        :
        MEND                        MACRO TRAILER
OPEN    START                       START OF OPEN CODE
        :
        MACID OPERAND1,OPERAND2     MACRO CALL
        :
        MACID OPERAND3,OPERAND4     MACRO CALL
        :
        END                         END OF SOURCE MODULE
```

You can call an inline macro definition only from the source module in which it is included. You can call a library macro definition from any source module. You can code a calling instruction anywhere in a source module, except before or between any inline macro definitions contained in that source module. You can also call a macro definition from within another macro definition. This type of call is an inner macro call; it is said to be *nested* in the macro definition.

Contents of a Macro Definition

The body of a macro definition can contain a combination of model statements, processing statements, and comment statements.

Model Statements

Model statements are assembler or machine instructions. As model statements, these instructions can use variable symbols as points of substitution. The macro assembler substitutes character string values in place of the variable symbols each time the macro is called.

The assembler processes the generated statements, with or without value substitution, at assembly time.

The 3 types of variable symbols in the assembler language are:

- Symbolic parameters, which are declared in the prototype statement
- System variable symbols
- SET symbols, which are part of the conditional assembly language

Processing Statements

Processing statements perform functions at preassembly time when macros are expanded, but they are not themselves generated for further processing at assembly time. The processing statements are:

- Conditional assembly instructions
- Inner macro calls
- MNOTE instructions
- MEXIT instructions

The MNOTE instruction allows you to generate an error message with an error condition code attached, or to generate comments in which you can display the results of preassembly operations.

The MEXIT instruction tells the assembler to stop processing a macro definition. The MEXIT instruction provides an exit from the middle of a macro definition. The MEND instruction not only delimits the contents of a macro definition, but also provides an exit from the definition.

The conditonal assembly language provides:

- Variables
- Data attributes
- Expression computation
- Assignment instructions
- Labels for branching
- Branching instructions
- Substring operators that select characters from a string

You can use the conditional assembly language in a macro definition to operate on input from a calling macro instruction. You can use the functions of the conditional assembly language (1) to select statements for generation, (2) to determine their order of generation, (3) to perform computations that affect the content of the generated statements, and (4) to produce preassembly messages through the MNOTE instruction. The conditional assembly language is fully described in this chapter.

Note. Conditional assembly instructions can be used only within macro definitions.

Comment Statements

A macro definition can contain two types of comment statements—one type describes preassembly operations and is not generated when the macro is expanded; the other type describes assembly operations and is generated. For details, see "Comment Statements in Macro Definitions." When a macro definition is called, the assembler generates assembler language statements.

Where To Place a Macro Definition in the Source Module

A macro definition within a source module must be at the beginning of that source module.

Open code is that part of a source module that is outside of and after any inline macro definition. Open code is initiated by any statement of the assembler language that appears outside of a macro definition, except the ICTL, ISEQ, EJECT, PRINT, SPACE, or TITLE instructions or a comment statement. Statements that do not start open code and comment statements can appear at the beginning of a source module:

- Before all macro definitions
- Between macro definitions
- After macro definitions and before open code

All other statements of the assembler language must appear after any inline macro definitions that are specified.

Parts of a Macro Definition

A macro definition consists of a header, prototype statement, body, and trailer.

Macro header. The MACRO instruction is the macro definition header; it must be the first statement of every macro definition. Its format is:

Name	*Operation*	*Operand*
blank	MACRO	blank

Prototype Statement. The prototype statement in a macro definition serves as a model (prototype) of a macro instruction used to call the macro definition. The prototype statement must be the second statement in every macro definition. It comes immediately after the MACRO instruction. The format of the prototype statement is:

Name	*Operation*	*Operand*
[label]	macro name	zero to 100 symbolic parameters, separated by commas

where *label* can be a symbolic parameter or blank.

Body of Macro. The machine instructions generated during macro expansion are determined by the machine, assembler, and conditional assembly instructions coded between the prototype statement and macro trailer.

Macro Trailer. The MEND instruction indicates the end of a macro definition. It also provides an exit when it is processed during macro expansion. Its format is:

Name	*Operation*	*Operand*
[label]	MEND	blank

where *label* can be a symbolic parameter or blank.

Coding the Prototype Statement

If no parameters are specified on the prototype statement, remarks are not allowed. Remarks are allowed after parameters, if preceded by at least one blank. To intersperse remarks with parameters (for example, a remark for each parameter), use continuation lines as shown below. Any number of continuation lines is allowed. However, each continuation line must be indicated by a nonblank character in the column after the end column on the preceding input record. For each continuation line, the symbolic parameters must begin in the begin column; otherwise, the whole line and any lines that follow are considered to contain remarks. See the example on the next page.

```
MOVE  &TO,                    REMARKS                          X
      &FROM,                  REMARKS                          X
      &LENGTH,                REMARKS                          X
      &PARAM,&PARAM2,&PARAM3, REMARKS                          X
      :
      &PARAM15                REMARKS
```

Prototype Label. You can write a parameter, similar to a symbolic parameter, as the label of a macro prototype statement. You can then assign a value to this parameter from the name entry in the calling macro instruction. If used, the label must be a variable symbol. If this parameter also appears in the body of a macro, it is given the value assigned to the parameter in the label of the corresponding macro instruction. For example:

```
* INSTRUCTIONS                          DESCRIPTION
*
          MACRO
&NAME     INTRCHG &TO,&FROM             PROTOTYPE STATEMENT
          :
&NAME     MVW   R1,SAVE
          MVW   &FROM,R1
          MVW   &TO,&FROM
          MVW   R1,&TO
          MVW   SAVE,R1
          :
          MEND
          START
          :
HERE      INTRCHG RESULT,DATA           MACRO CALL THAT GENERATES
          :                             FOLLOWING STATEMENTS:
          :
          :                             HERE MVW R1,SAVE
          :                                  MVW DATA,R1
          :                                  MVW RESULT,DATA
          :                                  MVW R1,RESULT
          :                                  MVW SAVE,R1
          :                                  :
          END
```

Note that the value assigned to the label parameter on the prototype statement has special restrictions that are listed in this chapter under "Using the Calling Macro Instruction."

Prototype Macro Name. The macro name is a symbol that identifies the macro definition. When you specify it in the operation field of a source instruction, the appropriate macro definition is called and processed by the assembler. The operation code specified in the prototype statement (prototype macro name) must not be the same as that specified in:

1. Any machine instruction.
2. Any assembler instruction other than the macro call.
3. The prototype statement of any other inline (or source file) macro definition. (If the name of a source file macro definition matches the name of one of your inline macro definitions, the assembler uses the inline definition.)

Prototype Symbolic Parameters. The operand entry in a prototype statement can contain positional or keyword symbolic parameters. These parameters represent the values passed from the calling macro instruction to the statements within the body of a macro definition. (See "Symbolic Parameters" in this chapter.)

Note. The operands must be symbolic parameters; parameters in sublists are not allowed. For a discussion of sublists in macro instruction operands, see "Sublists in Operands" in this chapter.

The Body of a Macro Definition

The body of a macro definition contains the sequence of statements that are the working part of a macro, including:

- Model statements to be generated
- Processing statements that, for example, can alter the content and sequence of the statements generated or issue error messages
- Comment statements, some of which are generated and others which are not
- Conditional assembly instructions to compute results to be displayed in the message created by the MNOTE instruction, without causing any assembler language statements to be generated

The statements in the body of a macro definition must appear between the macro prototype statement and the MEND statement.

Symbolic Parameters

Symbolic parameters (recognized by an ampersand as initial character) are declared in the macro prototype statement and serve as points of substitution in the body of the macro definition. During macro expansion, they are replaced by the values assigned to them by the calling macro instruction. By using symbolic parameters with meaningful names, you can indicate the purpose for the parameters (or substituted values).

Symbolic parameters must be valid variable symbols, consisting of an ampersand followed by an alphabetic character, followed by 0–6 alphameric characters (maximum of 8 characters total). They have a local scope; that is, the value they are assigned only applies to the macro definition in which they have been declared. The value of the parameter remains constant throughout each processing of the containing macro definition, changing with each call to the macro definition based on values assigned by each macro call.

Note. Symbolic parameters must not be defined in duplicate or be identical to any other variable symbols within the given local scope. (This applies to system variable symbols, and local and global SET symbols described later in this chapter.)

There are 2 kinds of symbolic parameters:

- Positional parameters; for example:
  Prototype: MYMAC & PARM1,& PARM2
  Calling macro: MYMAC FIELDA,FIELDB
- Keyword parameters; for example:
  Prototype: MYMAC2 & TO=,& FROM=
  Calling macro: MYMAC2 TO=FIELDA,FROM=FIELDB
  The two types of symbolic parameters may be mixed; for example:
  Prototype: MYMAC & PARAM1,& FROM=
  Calling macro: MYMAC FIELDA,FROM=FIELDB

All positional parameters must precede any keyword parameters, if the two kinds are mixed on a prototype statement.

If a parameter is positional on the prototype statement, it must be positional on the calling macro instruction; likewise, if keywords are used on the prototype statement, they must also be used on the calling macro instruction. Positional parameters on the calling macro instruction must appear in the same sequence as corresponding positional parameters on the prototype statement. Keyword parameters on the calling macro instruction do not have to appear in the same sequence as specified on the prototype statement, but must follow any positional parameters on the same calling macro instruction.

Which kind of parameters should you use—positional or keyword? There are advantages to each:

Positional. You should use a positional parameter if the value of the parameter changes with each calling macro instruction. Less coding is required to supply the value for a positional parameter than for a keyword parameter, since you code only the value; with keyword parameters, you must also code the keyword and equal sign.

Keyword. You should use keyword parameters if you have a large number of parameters. The keywords make it easier to identify which values are being assigned to which parameters on the calling macro instruction in any order. You should also use a keyword parameter if the value changes infrequently. Keyword parameters can be initialized to default values in the prototype statement; then if the calling macro instruction does not change that default value, it need not contain that parameter. For example:
Prototype: MYMAC & TO=FIELDA,& FROM=FIELDB
Calling macro: MYMAC TO=FIELDC

Values are assigned to keyword parameters as follows:

- If the corresponding keyword appears on the calling macro instruction, the value after the equal sign is the value for the parameter in that macro expansion.
- If the corresponding keyword does not appear on the calling macro instruction, the default value from the prototype statement is the value for the parameter in that macro expansion.

```
* INSTRUCTIONS                          DESCRIPTION
*
         MACRO
         KEYS  &KEY1=ABC,&KEY2=(A,B,C) PROTOTYPE STATEMENT
         :
         MEND
         START
         :
         KEYS                          MACRO CALL THAT GENERATES CODE
         :                             USING THE FOLLOWING VALUES:
         :
         :                                  &KEY1=ABC
         :                                  &KEY2=(A,B,C)
         :
         KEYS  KEY1=DEF,KEY2=(D,E,F) MACRO CALL THAT GENERATES CODE
         :                             USING THE FOLLOWING VALUES:
         :
         :                                  &KEY1=DEF
         :                                  &KEY2=(D,E,F)
         END
```

Note. A null character string can be specified as the default value of a keyword parameter and will be generated if the corresponding keyword operand is omitted.

```
* INSTRUCTIONS                          DESCRIPTION
*
         MACRO
         FXDPT &TYPE=,&REG=R3          PROTOTYPE STATEMENT--NULL CHARACTER
         :                             STRING DEFAULT VALUE FOR &TYPE
         :
         BAL&TYPE ADDR,&REG
         :
         B&TYPE ADDRESS
         MEND
OPEN     START
         :
         FXDPT                         MACRO CALL THAT GENERATES
         :                             THE FOLLOWING CODE:
         :
         :                                  BAL ADDR,R3
         :                                  :
         :                                  B   ADDRESS
         :
         FXDPT TYPE=X                  MACRO CALL THAT GENERATES
         :                             THE FOLLOWING CODE:
         :
         :                                  BALX ADDR,R3
         :                                  :
         :                                  BX   ADDRESS
         END
```

Subscripted Symbolic Parameters

Symbolic parameters may have several values expressed as a *sublist* rather than a single value. In this case, the symbolic parameter is written with a subscript, in the following format:

```
&PARAM( subscript )
```

where & PARAM is a valid variable symbol and *subscript* is an arithmetic expression (as described later in this chapter under "Arithmetic (SETA) Expressions"). The subscripted arithmetic expression can contain other subscripted variable symbols; nesting of subscripted variable symbols is allowed for up to five levels. The value of the subscript must be greater than or equal to 1.

The subscript on the prototype statement indicates the number of entries in the value sublist. The subscript in the body of the macro definition (when the symbolic parameter is used on a model statement) indicates the position of one entry in the sublist; if a symbolic parameter is subscripted on the prototype statement and nonsubscripted on a model statement, the model statement refers to the entire sublist (all entries). Sublists as values in calling macro instructions are fully described later in this chapter under "Sublists in the Macro Instruction Operand."

Model Statements

Assembler language instructions are generated from model statements at preassembly time. By specifying variable symbols as points of substitution in a model statement, you can vary the content of the instruction generated from that model statement.

A model statement consists of the same fields as an ordinary assembler language statement: name, operation, operand, and remarks. You cannot generate the identification and sequence field from a model statement. Model statements must have an entry in the operation field, in order to generate valid assembler language instructions. Each field or subfield can consist of:

- An ordinary character string
- A variable symbol as a point of substitution
- Any combination of ordinary character strings and variable symbols to form a concatenated string.

The statements generated at preassembly time from model statements must be valid machine or assembler instructions, and must not be conditional assembly instructions. They must obey the coding rules described in Chapter 2 or they will be flagged as errors at assembly time. A generated statement can occupy up to two continuation lines on the listing, unlike source statements, which are restricted to one continuation line.

Variable Symbols as Points of Substitution

Values can be substituted for variable symbols that appear in the name, operation, and operand fields of model statements; thus, variable symbols represent points of substitution. The three main types of variable symbols are:

- Symbolic parameters (positional and keyword)
- System variable symbols (& SYSLIST, & SYSNDX, & SYSPARM, & SYSDATE and & SYSTIME)
- SET symbols (global SETA, SETB, SETC and local SETA, SETB, SETC)

Symbolic parameters, SET symbols, and the system variable symbol & SYSLIST can all be subscripted. The remaining system variable symbols & SYSNDX, & SYSPARM, & SYSDATE, and & SYSTIME cannot be subscripted.

When values are substituted for variable symbols, the generated fields begin in standard columns, if possible.

Rules for Concatenation

When variable symbols are concatenated to ordinary character strings the following rules apply to the use of the concatenation character (a period).

- The concatenation character is mandatory when:
  - An alphameric character is to follow a variable symbol
  - A left parenthesis that does not enclose a subscript is to follow a variable symbol.
  - A period (.) is to be generated. Two periods must be specified in the concatenated string following a variable symbol.
- The concatenation character is not necessary when:
  - An ordinary character string precedes a variable symbol
  - A special character, except left parenthesis or period, follows a variable symbol
  - A variable symbol follows another variable symbol.

The concatenation character must not be used between a variable symbol and its subscript; otherwise, the characters will be considered a concatenated string and not a subscripted variable symbol.

Following are examples of concatenated strings and the resulting generated code:

Concatenated string	*Substituted values*	*Generated result*
&FIELD.A	&FIELD: AREA	AREAA
&FIELDA	&FIELDA: SUM	SUM
&DISP.(&BASE)	&DISP: 100 &BASE: 10	100 (10)
DC F'INT.. &FPACT'	&INT: 99 &FRACT: 88	DC F'99.88'
DC F'&INT&FRACT'	&INT: 99 &FRACT: 88	DC F'9988'
DC F'&INT.&FRACT'	&INT: 99 &FRACT: 88	DC F'9988'
FIELD&A	&A: A	FIELDA
&A + &B* 3 - D	&A: A &B: B	A + B* 3 - D
&A&B	&A: A &B: B	AB
&SYM(&SUBSCR)	&SUBSCR: 10 &SYM (10): ENTRY	ENTRY

Contents of Model Statement Name Field

The entries allowed in the name field of a model statement are:

- Blank
- Ordinary symbol
- Sequence symbol
- Variable symbol
- Any combination of variable symbols and other character strings concatenated

The name field of the generated statement must contain a valid ordinary symbol or blank. Variable symbols must not be used to generate comment statement indicators (an asterisk in the "begin" column).

Note. Restrictions on the name entry are further specified where each individual assembler language instruction is described in this manual.

Contents of Model Statement Operation Field

The entries allowed and not allowed in the operation field of a model statement are:

Allowed	*Not allowed*
• An ordinary symbol that represents the operation code for: –any machine instruction –a macro instruction –the following assembler instructions: ALIGN EJECT PRINT COM ENTRY PUSH COPY EQU SPACE CSECT EQUR START DC EXTRN TITLE DROP GLOBL USING DS ORG WXTRN DSECT POP • A variable symbol • A combination of variable symbols and other character strings concatenated together	• Blank • The assembler operation codes: END ICTL ISEQ MACRO

As a result, the entries allowed and not allowed in the operation field of the generated statements are:

Allowed	*Not allowed*
• An ordinary symbol that represents the operation code for: –any machine instruction –the following assembler instructions: ALIGN ENTRY PRINT COM EQU PUSH CSECT EQUR SPACE DC EXTRN START DROP GLOBL TITLE DS MNOTE USING DSECT ORG WXTRN EJECT POP	• Blank • Macro instruction operation code • A conditional assembly operation code: ACTR LCLA AGO LCLB AIF LCLC ANOP SETA GBLA SETB GBLB SETC GBLC • The following assembler operation codes: COPY ISEQ MEXIT END MACRO ICTL MEND

The MACRO and MEND operation codes are not allowed in model statements; they are used only for delimiting macro definitions. The END operation code is not allowed inside a macro definition.

Note. The MNOTE and MEXIT statements are not model statements. The MNOTE operation code can, however, be created by substitution.

Contents of Model Statement Operand Field

The entries allowed in the operand field of a model statement are:

- Blank (if valid)
- An ordinary symbol
- A character string combining alphameric and special characters (but not variable symbols)
- A variable symbol
- A combination of variable symbols and other character strings concatenated

The generated statement operand field must contain a blank or character string that represents a valid assembler or machine instruction operand.

Contents of Model Statement Remarks Field

Any combination of characters can be specified in the remarks field of a model statement. No values are substituted into variable symbols in this field.

Examples of Model Statements

```
Model:      &NAME     &OP    &REG,&ADDR          REMARKS &REG

Generated:  LABEL     MVW    R3,ADCON            REMARKS &REG
```

Note. Value is not substituted in remarks field.

```
                      LCLC   &ADDR
            &ADDR     SETC   'ADCON MA'
                      :
Model:                AW     R3,&ADDR  REMARKS

Generated:            AW     R3,ADCON MA REMARKS
```

Note. Space between ADCON and MA in the SETC model statement causes MA to be generated as part of the remarks field.

```
                      LCLC   &A
                      LCLC   &C
            &A        SETA   3
            &C        SETC   'R&A  &A'
                      :
Model:                CMR    &C  IS REGISTER COMPLEMENTED

Generated:            CMR    R3  3 IS REGISTER COMPLEMENTED
```

Note. Generated remarks are combined with remarks field of model statement.

```
            &STMT     SETC   'A  CMR   R3'
Model:      &STMT

Generated:            **ERROR**
```

Note. The generated statement has no operation field.

Processing Statements

The processing statements are:

- Conditional assembly instructions
- Inner macro instructions
- MNOTE instructions
- MEXIT instructions

Conditional Assembly Instructions

Conditional assembly instructions allow you to control at preassembly time the contents of the generated statements and the sequence in which they are generated. The instructions and their functions are:

Conditional assembly instruction	*Function*
GBLA, GBLB, GBLC LCLA, LCLB, LCLC	Declaration of initial value, type, and array dimensions for variable symbols (global and local SET symbols)
SETA, SETB, SETC	Assignment of values to variable symbols (SET symbols)
AIF	Conditional branch (based on logical test)
AGO	Unconditional branch
ANOP	Branch to next sequential instruction (no operation)
ACTR	Set loop counter

Inner Macro Instructions

Macro instructions can be nested inside macro definitions, allowing you to call other macros from within your own definitions. Nesting of macro instructions is fully described in "Nesting in Macro Definitions" in this chapter.

COPY Instruction

The COPY instruction, inside macro definitions, allows you to copy into the macro definition any sequence of statements allowed in the body of a macro definition. These statements become part of the body of the macro before macro processing takes place. You can also use the COPY instruction to copy complete macro definitions into a source module.

The specifications for the COPY instruction, which can also be used in open code, are described in Chapter 5, under "Program Sectioning."

MNOTE Instruction

You can use the MNOTE instruction to generate your own error messages or display intermediate values of variable symbols computed at preassembly time.

The MNOTE instruction is used inside macro definitions and its operation code can be created by substitution. The MNOTE instruction causes the generation of a message which is given a statement number in the printed listing.

The format of the MNOTE instruction statement is:

Name	*Operation*	*Operand*
[label]	MNOTE	message specification

The name field can contain a sequence symbol or blank. The message specification is 1 of 4 options:

Message specification	*Message produced*
n'message'	error message, severity n(0 – 255)
, 'message'	error message, severity 0
'message'	error message, severity 0
*, 'message'	comments, severity 0

The n stands for a severity code. The rules for specifying the contents of the severity code subfield are as follows:

- The severity code can be specified as a decimal self-defining term, or as a variable symbol representing a decimal self-defining term. The self-defining term must have a value in the range 0–255.
- If the severity code is omitted, with or without the comma, the assembler assigns a default value of 0 as the severity code.
- An asterisk in the severity code subfield causes the message and the asterisk to be generated as a comment statement.

The following examples show the four options for MNOTE operands:

- MNOTE 2,'ERROR IN SYNTAX'
  Generates severity 2 diagnostic error message.
- MNOTE ,'MISSING OPERAND'
  Generates severity 1 diagnostic error message.
- MNOTE 'INVALID PARAMETER'
  Generates severity 0 diagnostic error message.
- MNOTE *,'DEFAULT VALUE TAKEN'
  Generates a comment-type MNOTE

An MNOTE instruction causes a message to be printed if the current PRINT option is ON, even if the PRINT NOGEN option is specified.

Any combination of characters enclosed in apostrophes can be specified in the message subfield. The rules that apply to this character string are:

- Variable symbols are allowed (variable symbols can have a value that includes even the enclosing apostrophes).
- Double ampersands or double apostrophes are needed to generate one ampersand or one apostrophe. If variable symbols have ampersands or apostrophes as values, the values must have double ampersands or apostrophes.
- Any remarks for the MNOTE instruction statement must be separated from the apostrophe that ends the message by one or more blanks.
- Single apostrophes substituted or specified cause message generation to stop where the single apostrophe appears. If a single apostrophe is substituted in a position immediately after the closing apostrophe of the MNOTE instruction, then the apostrophe is printed. An error message is issued because a closing apostrophe cannot be found.

The following examples indicate the results generated during preassembly processing of MNOTE instructions:

MNOTE instruction	*Generated result*
MNOTE 3, 'THIS IS A MESSAGE'	3, THIS IS A MESSAGE
MNOTE 3, &PARAM (&PARAM = 'ERROR')	3, ERROR
MNOTE 3, 'VALUE OF &&A IS &A' (&A = 10)	3, VALUE OF &A IS 10
MNOTE 3, 'DOUBLE &AMPS' (&AMPS = &&)	3. DOUBLE &
MNOTE 3, 'DOUBLE &APOS' (&APOS = ' '	3, DOUBLE '
MNOTE 3, 'MESSAGE STOP' RMRKS	3, MESSAGE STOP RMRKS

MEXIT Instruction

The MEXIT instruction causes the assembler to exit from a macro definition to the next sequential instruction after the calling macro instruction. (This also applies to nested macro instructions.) Its format is:

Name	*Operation*	*Operand*
[label]	MEXIT	blank

where name is either a sequence symbol or blank.

Comment Statements In Macro Definitions

Macro definitions can contain two kinds of comment statements:

- Internal macro comments—used to describe operations performed at preassembly time; not generated in the macro expansion
- Ordinary comments—used to describe operations performed at assembly time; generated in macro expansion

No values are substituted for variable symbols specified in either internal or ordinary comments. If you want to display the value of a variable symbol, use a comment-type MNOTE.

The format of an internal macro comment is:

Column	*Contents*
1	period (.)
2	asterisk (*)
3–72	text of comment (any character string)

For example:

```
.*THIS IS AN INTERNAL MACRO COMMENT
```

The format for an ordinary comment statement within a macro definition is the same as for comment statements in open code (described in Chapter 2).

System Variable Symbols

There are five variable symbols whose values are set by the assembler according to specific rules; these are the system variable symbols:

- & SYSLIST—to refer to positional parameter or sublist in the calling macro instruction when there is no corresponding parameter or sublist in the prototype statement and to count the number of positional parameters or items in a positional parameter sublist
- & SYSNDX—to generate unique symbols for each expansion of a macro definition, by concatenating to the symbol a suffix whose value changes for each expansion
- & SYSPARM—to refer to a parameter specified in the assembler options list
- & SYSDATE—to provide the date of assembly
- & SYSTIME—to provide the time of the start of the assembly

You can use these symbols as points of substitution in model statements and conditional assembly instructions. All system variable symbols are subject to the same rules of concatenation and substitution as other variable symbols (see "Model Statements"). System variable symbols must not be used as symbolic parameters in the macro prototype statement. Also, they must not be declared as SET symbols. The assembler assigns read-only values to system variable symbols; they cannot be changed by using the SETA, SETB, or SETC instructions (see "Declaring SET Symbols").

The system variable symbols & SYSLIST and & SYSNDX are assigned a read-only value each time a macro is called and have that value only within that expansion of the macro. The system variable symbols & SYSPARM, & SYSDATE and & SYSTIME are assigned a read-only value for an entire source module.

&SYSLIST—Refer to Positional Parameters and Sublists

By varying the subscripts attached to & SYSLIST, you can refer to any positional parameter or sublist entry in a calling macro instruction. & SYSLIST can refer to positional parameters that have no corresponding positional parameter in the macro prototype statement. & SYSLIST can also count the number of positional parameters or entries in a positional sublist that were given on the calling macro instruction.

The assembler assigns read-only values to & SYSLIST each time a macro definition is called, applicable to that expansion of the macro only. & SYSLIST refers to the complete list of positional parameters in a calling macro instruction; & SYSLIST does *not* refer to *keyword* parameters.

When used as a point of substitution within the macro definition, one of 2 forms of & SYSLIST must be used:

- To refer to a positional parameter
  Calling macro instruction:
  ```
  MACLST P1,P2,...,Pn,...
  ```
  Point of substitution:
  ```
  & SYSLIST(n)
  ```
- To refer to a sublist entry in a positional parameter
  Calling macro instruction:
  ```
  MACSUB P1,P2,...,(Pn1,Pn2,...,Pnm,...),...
  ```
  Point of substitution:
  ```
  & SYSLIST(n,m)
  ```

The subscript n indicates the position of the parameter referred to. The subscript m, if specified, indicates the position of an entry in a sublist. The subscripts n and m can both be any arithmetic expression allowed in the operand of a SETA instruction; they must be greater than or equal to one.

If n refers to an omitted parameter or refers past the end of the complete list of positional parameters the null character string is substituted for & SYSLIST(n). If m refers to an omitted entry or refers past the end of the sublist, the null character string is substituted for & SYSLIST(n,m). Further, if the nth positional parameter is not a sublist, & SYSLIST(n,1) refers to the nth parameter and & SYSLIST(n,m) causes the null character string to be substituted if m is greater than one.

As an example of values substituted for & SYSLIST, consider the calling macro instruction:

```
MACALL ONE,TWO,(3,4,,6),,EIGHT
```

This results in the following value substitutions:

Point of substitution in macro definition	*Value substituted*
&SYSLIST (2)	TWO
&SYSLIST (3, 2)	4
&SYSLIST (4)	Null
&SYSLIST (9)	Null
&SYSLIST (3, 3)	Null
&SYSLIST (3, 5)	Null
&SYSLIST (2, 1)	TWO
&SYSLIST (2, 2)	Null
&SYSLIST (3)	(3, 4, , 6)

The attributes of the previously described forms of & SYSLIST are the attributes inherent in the positional parameter or sublist entry referred to.

There are two forms of & SYSLIST:

- To indicate the number of positional parameters in a macro call, use the form:
  N' & SYSLIST
- To indicate the number of sublist entries in a positional parameter, use the form:
  N' & SYSLIST(n)
  where n indicates the positional parameter.

For N' & SYSLIST, positional parameters are counted if specifically omitted (by specifying the comma that would normally have followed the omitted parameter). A sublist is counted as one parameter.

For N' & SYSLIST(n), sublist entries are counted if specifically omitted (by specifying the comma that would normally have followed the omitted entry). If the nth parameter is not a sublist, the value of N' & SYSLIST(n) is one; if the nth parameter is omitted, the value of N' & SYSLIST(n) is zero.

The following examples show values for N' & SYSLIST:

Macro instruction		*Value of N'&SYSLIST*
MACLST	1, 2, 3, 4	4
MACLST	A, B, , D, E	5
MACLST	, A, B, C, D	5
MACLST	(A, B, C) , (D, E, F)	2
MACLST		0
MACLST	KEY1 = A, KEY2 = B	0
MACLST	A, B, KEY1 = C	2

The following examples show values for N' & SYSLIST(n):

Macro instruction		*Value of N'&SYSLIST (2)*
MACSUB	A, (1, 2, 3, 4, 5) , B	5
MACSUB	A, (1, , 3, , 5), B	5
MACSUB	A, (, 2, 3, 4, 5) , B	5
MACSUB	A, B, C	1
MACSUB	A, , C	0
MACSUB	A, KEY = (A, B, C)	0
MACSUB		0

&SYSNDX—Generate Unique Symbols for Multiple Expansions

To generate a unique suffix for a symbol used in a macro definition for each expansion of that macro, concatenate & SYSNDX to the symbol. Although the same symbol is generated by two or more expansions (two or more calling macro instructions), the suffix provided by & SYSNDX produces unique symbols.

The assembler assigns & SYSNDX a read-only value each time a macro definition is expanded (for each calling macro instruction); this value is a 4-digit number, starting at 0001 for the first macro call and increased by 1 for each subsequent macro call (including nested macro calls).

& SYSNDX alone does not generate a valid symbol. It must be concatenated as the *suffix* to another symbol, and that symbol must not contain more than 4 characters (for a total of not more than 8 characters). For example, ITEM& SYSNDX. If & SYSNDX is concatenated to a variable symbol, the parameter assigned to that variable symbol must not contain more than 4 characters. For example, if the parameter THREE is substituted for the variable symbol & PRM& SYSNDX, the result would be THREEnnnn, which exceeds the length maximum for symbol names.

The type attribute of & SYSNDX, when used as a parameter on an inner macro call, is always N, and the count attribute is always 4.

The following example indicates the results of using & SYSNDX in naming DC and DS instructions.

```
* SOURCE CODE                          GENERATED CODE
*
         MACRO
         CONST &P1,&P2
         :
&P1&SYSNDX DC  F'&P2'
AREA&SYSNDX DS F
         MEND
OPEN     START
         :
         CONST TWO,2
         :                             TWOØØØ1   DC F'2'
         :                             AREAØØØ1  DS F
         CONST TWO,2ØØ
         :                             TWOØØØ2   DC F'2ØØ'
         :                             AREAØØØ2  DS F
         CONST THREE,3ØØ
         :                             THREEØØØ3 DC F'3ØØ'
         :                             AREAØØØ3  DS F
```

Example notes.

1. TWO0001 and TWO0002 are two different symbols, and thus are not multiply defined.
2. THREE0003 exceeds eight characters in length, causing an error.

&SYSPARM—System Parameter for Conditional Assembly

The system parameter & SYSPARM allows you to control conditional assembly flow and source code generation through the use of a parameter specified in the assembler options list. Thus, you can modify the output of an assembly without changing the source code itself.

The system parameter behaves like a global SETC symbol except that its value can be set only through the assembler options list. & SYSPARM cannot be modified during assembly and can only be coded inside macro definitions.

The system parameter contains the value of a character string within quotes, which must be zero to 8 characters long. It may consist of any combination of EBCDIC characters. A single quote in the string must be represented by two quotes. If no & SYSPARM value is specified, the value of the system parameter is a null string.

&SYSDATE—Date of Assembly

The value of the variable symbol is an 8-character string which is the date of the assembly. The format is mm/dd/yy (month/day/year) or dd/mm/yy (day/month/year) depending on which form was specified at SYSGEN. & SYSDATE cannot be modified during assembly and can only be coded inside macro definitions.

&SYSTIME—Time of Assembly

The value of the variable symbol is a 5-character string which provides the time at the start of the assembly. The format is hh.mm (hours.minutes). The value of & SYSTIME cannot be modified during assembly. For systems without the timer feature, & SYSTIME is a 5-character string of blanks.

Using the Calling Macro Instruction

The calling macro instruction (or macro call) provides the assembler with the name of a macro definition and the information or values you want passed to that macro definition. This information is the input to a macro definition. The assembler uses the information either in processing the macro definition or for substituting values into model statements during macro expansion. The output from a macro definition, called by a macro instruction, can be:

- A sequence of source statements generated from the model statements in the macro definition (macro expansion)
- Values assigned to global SET symbols, for use in other macro definitions

You can code a macro call anywhere in the open code part of your source module. However, the statements generated from the called macro definition must be valid assembler language instructions and allowed where the calling macro instruction appears. A macro call is not allowed before or between your inline macro definitions, but you can nest them inside a macro definition.

The format of a macro call statement is:

Name	*Operation*	*Operand*
[label]	macro name	zero to 100 operands, separated by commas

where the name field can contain any ordinary symbol or blank, and *macro name* identifies the macro definition to be expanded. The assembler allows up to 100 operands in the operand field. Your entries in the name, operation, and operand fields correspond to entries in the prototype statement of the called macro definition.

If you code no operands, remarks are not allowed. You can specify remarks on a macro instruction with operands in any of three ways:

1. The normal way, with all operands preceding all remarks; for example:

```
MACNORM PARM1,PARM2,PARM3,.....................................,X
      PARMN                   REMARKS
```

2. The alternate way, allowing remarks for each operand, with continuation lines used to pair remarks with parameters; for example:

```
MACALT PARM1,                 REMARKS ABOUT PARM1                  X
       PARM2,                 REMARKS ABOUT PARM2                  X
       PARM3,                 REMARKS ABOUT PARM3                  X
       :
       PARMN                  REMARKS ABOUT PARMN
```

3. A combination of the first two ways; for example:

```
MACOMB PARM1,PARM2,PARM3,  REMARKS                                 X
       PARM4,PARM5,        MORE REMARKS                            X
       :
       PARMN               MORE REMARKS
```

You are allowed any number of continuation lines. However, you must identify each continuation line with a nonblank character in the column after the end column of the previous statement line. Operands on continuation lines must begin in the continue column or beyond. If, in continuation lines, you make any entries in the columns preceding the continue column, the assembler issues an error message and does not process the entire statement.

Macro Instruction Name Field

The name field on a macro call can be used to generate an assembly-time label for a machine or assembler instruction. To accomplish this, a symbolic parameter must appear in the name field of the macro prototype statement and also in the name field of a model statement within the macro definition. Macro expansion will result in the name field of the model statement containing the name field entry from the calling macro instruction. See the example on the next page.

```
* SOURCE CODE                          GENERATED CODE
*
         MACRO
&NAM     MACNAM
         :
&NAM     MVWS  R6,(R2)
         :
         MEND
OPEN     START
         :
HERE     MACNAM
         :                                   :
         :                           HERE    MVWS R6,(R2)
         :                                   :
THERE    MACNAM
         :                                   :
         :                           THERE   MVWS R6,(R2)
         :                                   :
         END
```

Macro Instruction Operation Field

The symbolic operation code identifies the macro definition that you want the assembler to process. The operation entry for a macro instruction must be a valid symbol that is identical to the operation code in the prototype statement of the macro definition you want to call.

Note. If one of your inline macro definitions has the same name as a library macro definition, the assembler processes the inline macro definition.

Macro Instruction Operand Field

You use the operand entry in a macro instruction to pass values (parameters) to the called macro definition. These values can be passed through:

- The symbolic parameters you have specified in the macro prototype, or
- The system variable symbol & SYSLIST, if it is specified in the body of the macro definition.

Macro Instruction Operands

The assembler allows two types of parameters in a macro instruction operand; positional and keyword. You can specify a sublist with multiple values in both types of parameters. "Symbolic Parameters" earlier in this chapter explains the advantages of each type. Special rules for the various values passed in parameters are given in "Values in Macro Instruction Parameters" in this chapter.

Positional Parameters on the Macro Instruction

Use a positional parameter on the macro call to pass a value to a macro definition through the corresponding positional parameter declared on the prototype statement or to pass a value to the system variable symbol &SYSLIST.

If you specify &SYSLIST with appropriate subscripts in a macro definition, you do not need to declare positional parameters in the prototype statement. You can thus use &SYSLIST to refer to any positional parameter. And, &SYSLIST allows you to vary the number of parameters passed with each calling macro instruction.

If &SYSLIST is not used, you must code the positional parameters on the calling macro instruction in the same order and quantity as the positional parameters declared in the macro definition prototype statement. Otherwise:

- If the number of positional parameters on the calling macro instruction is greater than the number of positional parameters on the macro definition prototype statement, the excess parameters are meaningless.
- If the number of positional parameters on the calling macro instruction is less than the number of positional parameters on the macro definition prototype statement, the omitted parameters pass null character string values to corresponding parameters.

You must ensure that the nth parameter on the calling macro instruction and the nth parameter on the macro definition prototype statement are appropriately paired; omitted parameters on the calling macro instruction must be indicated specifically by coding the comma that would normally follow the omitted parameter, to maintain proper correspondence of subsequent parameters. For example:

```
* INSTRUCTIONS                        DESCRIPTION
*
         MACRO
         OMIT  &P1,&P2,&P3            PROTOTYPE STATEMENT
         DC    &P1&P2'ALWAYS &P3'     MODEL STATEMENT USING
         :                            SYMBOLIC PARAMETERS
         MEND
         START
         :
         OMIT  ,C,HERE                CALLING MACRO INSTRUCTION THAT
         :                            GENERATES THE FOLLOWING INSTRUCTION:
         :                                 DC C'ALWAYS HERE'
         END
```

Keyword Parameters on the Macro Instruction

Use a keyword parameter on the macro call to pass a value through a keyword parameter into a macro definition. To override the default value assigned to a keyword parameter on the prototype statement, code the corresponding keyword parameter on the macro instruction.

Any keyword parameter you specify in a macro instruction must correspond to a keyword parameter in the prototype statement. However, you do not have to code keyword parameters in any particular order.

You must code a keyword operand in the format *keyword=value*. The keyword coded on the calling macro instruction has up to seven characters and is *not* preceded by an ampersand; the corresponding keyword on the macro prototype statement consists of the same characters preceded by an ampersand. The value coded on the calling macro instruction can be up to 127 characters long. The value you specify overrides the default value in the prototype statement. The default value has the same rules as a value in a keyword parameter.

The following examples describe (1) the relationship between keyword operands and keyword parameters, and (2) the values that the assembler assigns to these parameters under different conditions.

```
* INSTRUCTIONS                          DESCRIPTION
*
         MACRO
         MAC   &KEY1=DEFAULT,&KEY2=  PROTOTYPE STATEMENT--DEFAULT VALUE
         :                           FOR &KEY2 IS A NULL CHARACTER STRING
SHOW     DC    C'&KEY1&KEY2'
         MEND
OPEN     START
         :
         MAC   KEY1=OVERRIDE,KEY2=1 CALLING MACRO INSTRUCTION THAT
         :                           GENERATES:
         :                           SHOW DC C'OVERRIDE1'
         :
         MAC   KEY1=OVERRIDE         CALLING MACRO INSTRUCTION THAT
         :                           GENERATES:
         :                           SHOW DC C'OVERRIDE'
         :
         MAC   KEY2=1                CALLING MACRO INSTRUCTION THAT
         :                           GENERATES:
         :                           SHOW DC C'DEFAULT1'
         :
         MAC                         CALLING MACRO INSTRUCTION THAT
         :                           GENERATES:
         :                           SHOW DC C'DEFAULT'
         END
```

The assembler issues an error message when the keyword of the calling macro instruction does not correspond to any keyword on the prototype statement.

You can specify the null character string as the value for a keyword parameter either as a default value on the prototype statement or on the calling macro instruction by coding *&keyword=* or *keyword=*, respectively, with no value following the equal sign. If another keyword parameter follows, the equal sign would be followed by a comma with no intervening blanks.

Combining Positional and Keyword Parameters

You can use a combination of positional and keyword parameters in the same macro instruction. See the following example.

```
* INSTRUCTIONS                          DESCRIPTION
*
         MACRO
         MIXED &P1,&P2,&P3=16,&P4=NO
         :
         :
         MEND
         START
         :
         MIXED 1Ø,YES                   CALLING MACRO INSTRUCTION THAT
         :                              USES POSITIONAL PARAMETERS
         :
         MIXED P3=1Ø,P4=YES             CALLING MACRO INSTRUCTION THAT
         :                              USES KEYWORD PARAMETERS
         :
         MIXED 1Ø,P4=YES                CALLING MACRO INSTRUCTION THAT
         :                              USES BOTH POSITIONAL AND
         :                              KEYWORD PARAMETERS
         :
         MIXED                          CALLING MACRO INSTRUCTION WITH NO
         :                              PARAMETERS--USING DEFAULT VALUES
         END
```

All positional parameters on the calling macro instruction must precede all keyword parameters on that instruction. The list of positional parameters must correspond to the positional parameters on the macro definition prototype statement as described under "Positional Parameters on the Macro Instruction."

Sublists in the Macro Instruction Operand

You can use a sublist in a positional or keyword parameter on the calling macro instruction to specify several values. A sublist is one or more entries separated by commas and enclosed in parentheses. The sublist, including parentheses, must not exceed 127 characters.

In a macro definition, you can refer to the value of each entry by coding:

- The corresponding symbolic parameter with an appropriate subscript, or
- The system variable symbol & SYSLIST with appropriate subscripts, the first to refer to the positional parameter and the second to refer to the sublist entry in the operand.

These rules apply to sublists in macro instructions:

- & SYSLIST can refer only to sublists in positional parameters.
- The value in a positional or keyword parameter can be a sublist.
- A symbolic parameter as used within the macro definition can refer to the entire sublist or to an individual entry of the sublist. To refer to an individual entry, the symbolic parameter must have a subscript whose value indicates the position of the entry in the sublist. The subscript must have a value greater than or equal to one.

The following shows an example of a sublist:

```
* INSTRUCTIONS                          DESCRIPTION
*
         MACRO
         MAC   &P1,&P2,&KEY=(F,Ø)       PROTOTYPE STATEMENT WITH SUBLIST
         :                              DEFAULT VALUE FOR KEYWORD PARAMETER
         :
KEY      DC    &KEY(1)'&KEY(2)'         MODEL STATEMENT--SUBSCRIPTS REFER TO
         :                              POSITIONS WITHIN SUBLIST VALUE FOR
         :                              &KEY
         :
&P1(1)   DC    &P1(2)'&P1(3)'           MODEL STATEMENT--SUBSCRIPTS REFER TO
         :                              POSITIONS WITHIN SUBLIST VALUE TO BE
         :                              PASSED AS POSITIONAL PARAMETER ON
         :                              MACRO CALL
         DC    A&P2
         :
         MEND
         START
         :
         MAC   (POS,F,2ØØ),(A,B,C)      MACRO CALL THAT GENERATES:
         :                              KEY DC F'Ø'
         :                                  :
         :                              POS DC F'2ØØ'
         :                                  :
         :                                  DC A(A,B,C)
         END
```

The following table shows the relationship between subscripted parameters and sublist entries when:

- A sublist entry is omitted.
- The subscript refers past the end of the sublist.
- The value of the operand is not a sublist.
- The parameter is not subscripted.

Parameter in macro definition	*Sublist in macro call*	*Value generated*
&PAR (3)	(1, 2, , 4)	Null character string
&PAR (5)	(1, 2, 3, 4)	Null character string
&PAR	A	A
&PAR (1)	A	A
&PAR (2)	A	Null character string
&PAR	(A)	(A) See Note 1 below.
&PAR (1)	(A)	A See Note 1 below.
&PAR (2)	(A)	Null character string See Note 1 below.
&PAR	( )	( ) See Note 2 below.
&PAR (1)	( )	( ) See Note 2 below.
&PAR (3)	( )	Null character string See Note 2 below.
&PAR (2)	(A, , C, D)	Nothing See Note 3 below.
&PAR (1)	( )	Nothing See Note 4 below.

Note 1. Because the single value A is enclosed in parentheses, it is considered a sublist with one entry.

Note 2. The value of the operand is not a sublist. It is considered to be a character string.

Note 3. The blank between commas on the calling macro instruction indicates end of the operand field; thus instead of a null character string value being passed to the macro definition, no value is passed. In addition, an error message is generated because the assembler considers this an unmatched left parenthesis.

Note 4. The blank following the left parenthesis indicates end of the operand field; thus instead of a null character string value being passed to the macro definition, no value is passed. In addition, an error message is generated because the assembler considers this an unmatched left parenthesis.

The macro definition can also use & SYSLIST to refer to sublist entries in positional parameters. For example, given the sublist
A,(1,2,3,4)
& SYSLIST(2,3) would have the value 3.

Values in Macro Instruction Parameters

You can use a macro instruction parameter to pass values to a macro definition. The two types of values you can pass are:

- Explicit values, or the actual character strings you specify in the operand of the calling macro instruction
- Implicit values, or the attributes inherent to the data represented by the explicit values

The explicit value of a macro instruction parameter is a character string that must not exceed a length of 127 characters (including any sublists). If the macro call is contained within a macro definition (a nested macro call), the explicit value of the parameter may contain:

- Variable symbols
- Any of the symbolic parameters specified in the prototype statement of the containing macro definition
- Any SET symbols declared in the containing macro definition
- The system variable symbols

If the macro call is in open code, it cannot contain the above symbols.

The assembler assigns the character string value, including sublist entries, to the corresponding parameter declared in the prototype statement. A sublist entry is assigned to the corresponding subscripted parameter. When you omit a keyword parameter on the calling macro instruction, the assembler assigns the default value specified for the corresponding keyword parameter on the prototype statement. When you omit a positional parameter or sublist entry, on the calling macro instruction, the assembler assigns the null character string to the parameter. Any of the 256 characters of the EBCDIC character set can appear in a macro instruction parameter (or sublist entry). However, the following characters require special consideration:

Ampersands. In macro calls nested within macro definitions a single ampersand indicates the presence of a variable symbol. The assembler substitutes the value of the variable symbol into the character string specified in a macro instruction parameter. The resultant string is then the value passed into the macro definition. If the variable symbol is undefined, the assembler issues an error message. You must specify double ampersands if they are to be passed without substitution to the macro definition.

Value specified on macro call	*Value of variable symbols*	*Character string values passed*
&VAR	XYZ	XYZ
&A + &B + 3 + &C* 10	&A = 2 &B = X &C = COUNT	2 + X + 3 + COUNT* 10
'&MESSAGE' (see *Quoted Strings* in this list.)	BLANK BETWEEN	'BLANK BETWEEN'
&&REGISTER		&&REGISTER
NOTE&&&&		NOTE&&&&

Apostrophes. A single apostrophe indicates the beginning and end of a quoted string.

Quoted Strings. A quoted string is any sequence of characters that begins and ends with a single apostrophe (compare with conditional assembly character expressions). You must specify double apostrophes inside each quoted string to result in a single apostrophe value. This includes substituted apostrophes. Macro instruction parameters can have values that include one or more quoted strings. Each quoted string can be separated from the following quoted string by one or more characters, and each must contain an even number of apostrophes. Quoted strings can contain variable symbols only on macro calls nested within macro definitions. The following examples indicate the values passed for quoted strings.

Value on macro call	*Value of variable symbol*	*Value passed*
'&&NOTATION'		'&&NOTATION'
'&MESSAGE'	&MESSAGE = OK	'OK'
''		''
'&QUOTES'	&QUOTES='' ''	''' '''
'QUOTE1' AND 'QUOTE2'		'QUOTE1' AND 'QUOTE2'
'QUOTE1' 'QUOTE2'		'QUOTE1' 'QUOTE2'

Parentheses. In macro instruction operand values, there must be an equal number of left and right parentheses. They must be paired; that is, to each left parenthesis belongs a following right parenthesis at the same level of nesting. An unpaired (single) left or right parenthesis can appear only in a quoted string.

Blanks. One or more blanks outside a quoted string indicates the end of the entire operand field in a macro instruction. Thus, blanks should be used only inside quoted strings.

Commas. A comma outside a quoted string indicates the end of a parameter value or sublist entry. Commas that do not delimit values can appear inside quoted strings or inside paired parentheses that do not enclose sublists. A comma must not follow the final parameter specification.

Equal Signs. An equal sign can appear in the value of a macro instruction parameter sublist entry (1) inside quoted strings, or (2) between paired parentheses. For example:

```
MACCALL A'='B,C(A=B)
```

Where '=' is a quoted string and C(A=B) is a character string.

Periods. A period can appear in the value of a parameter or sublist entry. It will be passed as a period. However, if you use a period immediately after a variable symbol (valid only on macro calls from within macro definitions), it becomes a concatenation character. In this case, you must code two periods if one is to be passed as a character.

Character string on macro call	*Value of variable symbol*	*Value passed*
3.4		3.4
(3.4, 3.5, 3.6)		(3.4, 3.5, 3.6)
&A.1	FIELD	FIELD1
&A.1	3	31
&A. . 1	3	3.1
&A&B	&A = AREA &B = 200	AREA200
&A.&B	&A = AREA &B = 200	AREA200
&DISP. (&BASE)	&DISP = 1000 &BASE = 10	1000 (10)

Nesting Macro Definitions

A nested macro instruction is a macro instruction specified as one of the statements in the body of a macro definition. The assembler allows the expansion of a macro definition from within another macro definition. Any macro instruction in the open code of a source module is an *outer macro call.* Any macro instruction that appears within a macro definition is an *inner macro call.* For example:

```
        MACRO
        OUTER                   PROTOTYPE STATEMENT
        :
        INNER                   INNER MACRO CALL
        :
        MEND
        MACRO
        INNER                   PROTOTYPE STATEMENT
        :
        MEND
OPEN    START Ø
        :
        OUTER                   OUTER MACRO CALL
        :
        END
```

Levels of Nesting

The code generated by a macro definition called by an inner macro call is nested inside the code generated by the macro definition that contains the inner macro call. In the macro definition called by an inner macro call, you can include a macro call to another macro definition. Thus, you can nest macro calls at different levels.

The zero level includes outer macro calls that appear in open code; the first level of nesting includes inner macro calls that appear inside macro definitions called from the zero level; the second level of nesting includes inner macro calls inside macro definitions that are called from the first level, etc.

You can also call a macro definition recursively—the body of a macro definition can contain an inner macro call to that same macro definition. In other words, a macro can call itself. This allows you to define macros to process recursive functions.

When macro instructions appear inside macro definitions, the assembler substitutes values in the same way as it does for the model statements in the containing macro definition. The assembler processes the called macro definition, passing to it the operand values (after substitution) from the inner macro instruction.

The number of nesting levels permitted depends on the complexity and size of the macros at the different levels: the number of operands you specify, the number of local and global SET symbols you declare, and the number of sequence symbols you use.

Note. Nesting macros may cause assembler performance to be slower.

Exits taken from the different levels of nesting when a MEXIT or MEND instruction is encountered are as follows:

- From the expansion of a macro definition called by an inner macro call, the exit is to the next sequential instruction after the inner macro call.
- From the expansion of a macro definition called by an outer macro, the exit is to the next sequential instruction after the outer macro call in your open code.

You can pass a parameter value in an outer macro instruction through one or more levels of nesting. However, the value you specify in the inner macro instructions must be identical to the corresponding symbolic parameter declared in the prototype statement. Thus, you can pass and refer to a sublist in the macro definition called by the inner macro call. Also, all symbols carry their inherent attribute values through the nesting levels. You can pass values from open code through several levels of macro nesting if you specify inner macro calls at each level with symbolic parameters as parameter values. For example:

```
* INSTRUCTIONS                    DESCRIPTION
*
      MACRO
      OUTER &P                    PROTOTYPE STATEMENT FOR OUTER
      :
      INNER &P                    NESTED MACRO CALL
      :
      MEND
      MACRO
      INNER &Q                    PROTOTYPE STATEMENT FOR INNER
      :
      MVW   &Q(1),R1
      AW    &Q(2),R1
      MVW   R1,&Q(3)
      :
      MEND
      START Ø                     START OF OPEN CODE
      :
      OUTER (AREA,F2ØØ,SUM)       MACRO CALL IN OPEN CODE PRODUCES THE
      :                           FOLLOWING NESTED MACRO CALL:
      :                                INNER (AREA,F2ØØ,SUM)
      :                           WHICH PRODUCES THE FOLLOWING:
      :                                MVW AREA,R1
      :                                AW  F2ØØ,R1
      :                                MVW R1,SUM
      END
```

Note. If a symbolic parameter is only a part of the value in an inner macro instruction, only the character string value given to the parameter by an outer call passes through the nesting level. Inner sublist entries and attributes of symbols are not available for reference in the inner macro.

```
*  INSTRUCTIONS                    DESCRIPTION
*
         MACRO
         OUTER &P,&Q                PROTOTYPE STATEMENT FOR OUTER
         :
         INNER (ABC,&P,DEF),&Q+3    NESTED MACRO CALL
         MEND
         MACRO
         INNER &R,&S                PROTOTYPE STATEMENT FOR INNER
         :
         DC    A&R(2)
         DS    XL(&S)
         MEND
         START 0                    START OF OPEN CODE
         :
         OUTER (ADX,ADY,ADZ),TWO0   MACRO CALL FROM OPEN CODE PRODUCES
         :                          THE FOLLOWING NESTED MACRO CALL:
         :
         :                          INNER (ABC,(ADX,ADY,ADZ),DEF),TWO0+3
         :
         :                          WHICH RESULTS IN:
         :                               DC A(ADX,ADY,ADZ)
         :                               DS XL(TWO0+3)
         END
```

The assembler gives system variable symbols local read-only values that depend on the position of your macro call and the parameter value specified on the macro call.

&SYSLIST. If you specify & SYSLIST in a macro definition called by an inner macro instruction, then & SYSLIST refers to the positional operands of the inner macro instruction.

&SYSNDX. The assembler increases the value of & SYSNDX by one each time it encounters a macro call. It retains the increased value throughout the expansion of the macro definition that is called; that is, within the local scope of the nesting level. For example, if open code contains a macro call to the macro INNER1, and INNER1 contains a call to the macro INNER2, and the value of & SYSNDX is 0001 when the call to INNER 1 is executed, then & SYSNDX will have the value 0002 throughout the expansion of INNER1 except for the portion of code included from INNER2 at the point of the call from INNER1 to INNER2 (for that portion & SYSNDX will have the value 0003); note that the value of & SYSNDX is 0002 for all code expanded from INNER1 after the portion inserted from INNER2.

&SYSPARM, &SYSDATE, &SYSTIME. The nesting of macros does not affect & SYSPARM, & SYSDATE, or & SYSTIME.

Conditional Assembly Language

This section describes the conditional assembly language used to interact with symbolic parameters and system variable symbols inside a macro definition. With the conditional assembly language, you can perform general arithmetic and logical computations as well as many of the other functions you can perform with any other programming language. By combining conditional assembly instructions with assembler and machine instructions, you can:

- Select sequences of model statements, from which the assembler generates machine and assembler instructions
- Vary the contents of these model statements during generation

The assembler processes the instructions and expressions of the conditional assembly language at preassembly time. Then, at assembly time, it processes the generated model statements. Conditional assembly instructions, however, are not processed after preassembly time.

The elements of the conditional assembly language are:

- SET symbols that represent data
- Attributes that represent different characteristics of data
- Sequence symbols that act as labels for branching to statements at preassembly time

The functions of the conditional assembly language are:

- Declaring SET symbols as variables for use by the conditional assembly language in its computations
- Assigning values to the declared SET symbols
- Evaluating conditional assembly expressions used as values for substitution, as subscripts for variable symbols, or as condition tests for branch instructions
- Selecting characters from strings, for substitution in and concatenation with other strings, or for inspection in condition tests
- Branching and exiting from conditional assembly loops

SET Symbols

SET symbols are variable symbols that provide arithmetic, binary, or character data, whose values you can vary at preassembly time.

You can use SET symbols as:

- Terms in conditional assembly expressions
- Counters, switches, and character strings
- Subscripts for variable symbols
- Values for substitution

Thus, SET symbols allow you to control your conditional assembly logic and to generate many different statements from the same model statements.

You can use a SET symbol to represent an array of many values. You can then refer to any one of the values in this array by subscripting the SET symbol.

You must declare a SET symbol before you can use it. If you declare a SET symbol to have a local scope, you can use it only in the statements that are part of the same macro definition. If you declare a SET symbol to have a global scope, you can use it in statements that are part of:

- The same macro definition
- Any other macro definition

You must, however, declare the SET symbol as global in each macro definition where you use it. You can change the value you previously assigned to a SET symbol without affecting the scope of the symbol.

Note. A symbolic parameter has a local scope. You can use it only in the statements that are part of the macro definition where you declare the parameter in the prototype statement. The values of &SYSNDX and &SYSLIST are local to each individual macro definition. The value of &SYSPARM is global—the same within all macro definitions.

Three types of SET symbols are used in model statements and conditional assembly instructions:

- SETA (arithmetic data)
- SETB (binary data)
- SETC (character data)

You must declare a SET symbol, to determine its scope and type, before you can use it. The declaration instructions for SET symbols are:

Instruction	*SET symbol type*	*Scope of SET symbol*
LCLA &symbol	SETA	Local
LCLB &symbol	SETB	Local
LCLC &symbol	SETC	Local
GBLA &symbol	SETA	Global
GBLB &symbol	SETB	Global
GBLC &symbol	SETC	Global

where *&symbol* indicates the name of the SET symbol declared.

Once a SET symbol has been declared, you can change the value of that symbol with the SETA, SETB or SETC instruction anywhere within the declared scope of the SET symbol. Values of symbolic parameters and system variable symbols, in contrast, remain fixed throughout their scope. Wherever a SET symbol appears in a statement, the assembler replaces the symbol with the last value assigned to the symbol.

The features of SET symbols are compared with those of symbolic parameters and system variable symbols in the following table.

	SET symbols	*Symbolic parameters*	*System variable symbols*
Local scope	yes	yes	yes
Global scope	yes	no	no–&SYSNDX, &SYSLIST yes–&SYSPARM, &SYSDATE, &SYSTIME
Values can change within scope	yes	no	no

Note. You can use SET symbols in the name and operand fields of inner macro calls; however, the assembler considers the value thus passed through a symbolic parameter into a macro definition to be a character string and generates it as such.

A subscripted SET symbol is written as:
&symbol(subscript)
where *&symbol* is a valid SET symbol name and the *subscript* is an arithmetic expression with a value greater than zero. You can use a subscripted SET symbol anywhere an unsubscripted SET symbol is allowed. However, you must declare subscripted SET symbols as dimensioned in a previous local or global declaration instruction. The subscript refers to one of the many positions in an array of values identified by the SET symbol. The value of the subscript must not exceed the dimension declared for the array in the corresponding local or global declaration instruction.
For example:

```
         LCLA  &ARRAY(20)          DECLARE ARRAY OF 20 ELEMENTS
          :
&ARRAY(5) SETA 5                   REFER TO FIFTH ELEMENT
```

Note. A subscript can be a subscripted SET symbol. Five levels of subscript nesting are allowed.

Data Attributes

Macro instruction operands can be described in terms of:

- *Type*, which distinguishes numeric data and identifies missing operands.
- *Count*, which gives the number of characters required to represent data in a macro instruction operand.
- *Number*, which gives the number of operands in a macro instruction or the number of sublist entries in an operand.

These three characteristics are called *data attributes*. The assembler assigns attribute values to the symbolic parameters and & SYSLIST that represent the operands.

Specifying attributes in conditional assembly instructions allows you to control conditional assembly logic, which in turn can control the sequence and contents of the statements generated from model statements. The specific purpose for which you use an attribute depends on the kind of attribute being considered. The attributes and their main uses are listed in the following table.

Attribute	*Purpose*	*Main uses*
Type	Gives a letter that identifies type of data represented	–In tests to distinguish numeric data –To discover missing operands
Count	Gives the number of characters required to represent data	–For scanning and decomposing of character strings –As indexes in sub-string notation
Number	Gives the number of sublist entries in operand sublist or number of operands in a macro instruction	–For scanning sublists –As counter to test for end of sublist

The format for an attribute reference is:
code'symbol
where *code* is one of the three attribute codes (T for type, K for count, or N for number) and *symbol* is a variable symbol; the apostrophe between *code* and *symbol* must be present.

The attribute notation indicates the attribute whose value you desire. The variable symbol represents the data that has the attribute. The assembler substitues the value of the attribute for the attribute reference. The attribute reference can appear only in conditional assembly instructions. Thus, their values are available only at preassembly time.

Note. You can use the system variable symbol & SYSLIST in an attribute reference to refer to a macro instruction operand.

Type Attribute (T)

The type attribute has a value of a single alphabetic character that indicates the type of data represented by a macro instruction operand. If the operand is a sublist, the entire sublist and each entry in the sublist can possess the type attribute.

Type attribute codes	*Data characterized by type attribute code*
N	A self-defining term used as macro instruction operand
O	Omitted macro instruction operand (has a value of a null character string)
U	Non-numeric macro instruction operand

You can use a type attribute reference only in the SETC instruction or as one of the comparison values in a SETB or AIF instruction.

Count Attribute (K)

The count attribute applies only to macro instruction operands. It has a numeric value equal to the total number of characters in a macro instruction operand. You can use a count attribute reference only in arithmetic expressions. The count attribute for an omitted operand is zero. For example:

Macro instruction operands	*Value of count attribute*
ALPHA	5
(SUB, LIST, ALL)	14
2 (10, 12)	8
'A' 'B'	6
' ' blank	3
'' null character string	2
(omitted operand)	0

Number Attribute (N)

The number attribute applies only to the operands of macro instructions. It has a numeric value that is equal to the number of sublist entries in an operand (1 + the number of commas separating the entries).

You can use a number attribute reference only in arithmetic expressions. N' & SYSLIST refers to the total number of positional parameters in a macro instruction, and N' & SYSLIST(m) refers to the number of sublist entries in the mth parameter. If the mth parameter is not a sublist, the value of N' & SYSLIST(m) is 1. For example:

Macro instruction operand sublist	*Value of number attribute*
(A, B, C, D)	4
(A, , B, C, D, E)	6
(, B, C, D)	4
(A)	1
A	1
(No operands)	0

Sequence Symbols

You can use a sequence symbol in the name field of a statement to branch to that statement at preassembly time, thus altering the sequence in which the assembler processes your conditional assembly and macro instructions. You can select the model statements from which the assembler generates assembler language statements.

A sequence symbol is written as a period followed by an alphabetic character, followed by 0–6 alphameric characters (for a total of 2–8 characters). For example:

```
.SEQ
.A1234
.#924
```

You can specify sequence symbols in the name field of any model statements in a macro definition, except instructions that already contain ordinary or variable symbols in the name field. You cannot specify a sequence symbol in the name field of the macro prototype statement.

You can specify sequence symbols in the operand field of an AIF or AGO instruction to branch to a statement with the same sequence symbol in its name field. Sequence symbols have a local scope. Thus, if you code a sequence symbol in an AIF or AGO instruction, you must define that sequence symbol as a label in the same macro definition. And, since the scope for sequence symbols is local, you can use the same sequence symbol in several macro definitions, without conflict. For example:

```
          MACRO
          MACONE
          :
          AGO    .GENERAT
          :
.GENERAT  ANOP
          :
          MEND
          MACRO
          MACTWO
          :
.GENERAT  ANOP
          :
          AGO    .GENERAT
          :
          MEND
OPEN      START
          :
          MACONE
          :
          MACTWO
          :
          END
```

Note. The assembler does not substitute a sequence symbol from the name field of an inner macro call for the parameter in the name field of the corresponding prototype statement.

Declaring SET Symbols

You must declare a SET symbol before you can use it. In the declaration, you specify whether it is to have a global or local scope. The assembler assigns an initial value to a SET symbol at its point of declaration. All global declarations must immediately follow the macro prototype statement; local declarations must immediately follow any global declarations.

LCLA, LCLB, and LCLC Instructions

The LCLA, LCLB, and LCLC instructions declare local SETA, SETB, and SETC symbols. The format of the LCLA, LCLB, and LCLC instructions is:

Name	*Operation*	*Operand*
blank	LCLA LCLB LCLC	one or more variable symbols to be used as SET symbols, separated by commas

The name field of the LCLA, LCLB, and LCLC instructions must be blank.

These instructions must appear immediately following any GBLA, GBLB, or GBLC instructions. Any variable symbols declared in the operand field have a local scope. You can use them as SET symbols anywhere after the pertinent LCLA, LCLB, or LCLC instructions, but only within the declared local scope. The following example indicates the scope of several local SET symbols:

```
         MACRO
         MAC1
         LCLA  &A1                SCOPE OF &A1 STARTS HERE
         LCLC  &C1                SCOPE OF &C1 STARTS HERE
         :
         MEND                     SCOPE OF &A1 AND &C1 ENDS HERE
         MACRO
         MAC2
         LCLA  &A2                SCOPE OF &A2 STARTS HERE
         LCLC  &C2                SCOPE OF &C2 STARTS HERE
         :
         MEND                     SCOPE OF &A2 AND &C2 ENDS HERE
OPEN     START
         :
         MAC1
         :
         MAC2
         :
         END
```

The assembler assigns initial values to local SET symbols as follows:

Instruction	*Initial value assigned to SET symbols*
LCLA	0
LCLB	0
LCLC	Null character string

A local SET variable symbol declared by the LCLA, LCLB, or LCLC instruction must not be identical to any other variable symbol within the same local scope. The following rules apply to a local SET variable symbol:

- It must not be the same as any symbolic parameter declared in the prototype statement.
- It must not be the same as any global variable symbol declared within the same local scope.
- The same variable symbol must not be declared or used as 2 different types of SET symbols, for example, as a SETA and a SETB symbol, within the same local scope.

Note. A local SET symbol should not begin with the characters & SYS; this prefix is reserved for system variable symbols.

You declare a subscripted local SET symbol with the LCLA, LCLB, or LCLC instruction by following the subject name with dimension information enclosed in parentheses. For example:

```
LCLA  &ARRAY(10)
```

The dimension must be an unsigned decimal self-defining term not equal to zero. The maximum dimension allowed is 255. The dimension indicates the number of SET variables associated with the subscripted SET symbol. The assembler assigns an initial value to every variable in the array (the same initial value as for nonsubscripted local SET symbols). You can use a subscripted local

SET symbol only if the declaration has a subscript, which represents a dimension. You can use a nonsubscripted local SET symbol only if the declaration has no subscript.

GBLA, GBLB, and GBLC Instructions

GBLA, GBLB, and GBLC instructions declare global SETA, SETB, and SETC symbols. The format of the GBLA, GBLB, and GBLC instruction statements is:

Name	*Operation*	*Operand*
blank	GBLA GBLB GBLC	one or more variable symbols to be used as SET symbols, separated by commas

The name field of the GBLA, GBLB, and GBLC instructions must be blank.

The GBLA, GBLB, and GBLC instructions must appear immediately following the macro prototype statement. Any variable symbols declared in the operand fields of these instructions have a global scope. You can use them as SET symbols anywhere after the pertinent GBLA, GBLB, or GBLC instructions; note that they can be used only in those macro definitions which contain the global declarations. Global scope means the value can be passed from one macro definition to another, as opposed to local scope which means the value is initialized for each macro definition.

For example, in the following code, values can pass between the macro definitions MAC1 and MAC2 through the global SET symbols & B and & C; the value of & A in MAC1 is unknown to MAC2, and the value of & A in MAC2 is unknown to MAC1 since & A is a local SET symbol.

```
        MACRO
        MAC1
        GBLB  &B
        GBLC  &C
        LCLA  &A
        :
        MEND
        MACRO
        MAC2
        GBLB  &B
        GBLC  &C
        LCLA  &A
        :
        MEND
OPEN    START
        :
        MAC1
        :
        MAC2
        :
        END
```

The assembler assigns an initial value to a global SET symbol when processing the first GBLA, GBLB, or GBLC instruction containing the symbol (in the first macro definition which declares the symbol to be a global SET symbol); initial values are not reassigned when the global SET symbol is subsequently declared in other macro definitions. The initial values assigned to global SET symbols are:

Instruction	*Initial value assigned to SET symbols*
GBLA	0
GBLB	0
GBLC	Null character string

The following example shows the values of a global SET symbol:

```
          MACRO
          FIRST
          GBLA  &A                    &A INITIALIZED TO Ø
          :
          MEND
          MACRO
          SECOND
          GBLA  &A
          :
          MEND
OPEN      START
          :
          FIRST                       &A=Ø
          :
          SECOND                      &A=VALUE DETERMINED BY FIRST
          :
          FIRST                       &A=VALUE DETERMINED BY SECOND
          :
          END
```

A global SET symbol declared by the GBLA, GBLB, or GBLC instruction must not be identical to any other variable symbol used within the same macro definition. The following rules apply to a global SET symbol:

- It must not be the same as any symbolic parameter declared in the prototype statement.
- It must not be the same as any local variable symbol declared within the same macro definition.
- The same variable symbol must not be declared or used as 2 different types of global SET symbol, for example, as SETA and SETB symbols.

Note. A global SET symbol should not begin with the characters & SYS; this prefix is reserved for system variable symbols.

You declare a subscripted global SET symbol with the GBLA, GBLB, or GBLC instruction by following the symbol name with dimension information enclosed in parentheses. For example:

```
GBLA  &ARRAY(1Ø)
```

The dimension must be an unsigned decimal self-defining term not equal to zero. The maximum dimension allowed is 255. The dimension indicates the number of SET variables associated with the subscripted SET symbol. The assembler assigns an initial value to every variable in the array (the same initial value as for nonsubscripted global SET symbols). You can use a subscripted global SET symbol only if the declaration has a subscript, which represents a dimension. You can use a nonsubscripted global SET symbol only if the declaration has no subscript. Wherever you declare a particular global SET symbol with a dimension as a subscript, the dimension must be the same in each declaration.

Assigning Values to Set Symbols

SETA—Assign Arithmetic Value

SETA assigns an arithmetic value to a SETA symbol. You can specify a single value or an arithmetic expression; the assembler will compute the value of the expression. Since you can change the value of a SETA symbol by assigning arithmetic expressions, you can use SETA symbols as counters, indexes, or for other repeated computations that require varying values.

The format of the SETA instruction is:

Name	*Operation*	*Operand*
symbol	SETA	arithmetic expression

The symbol in the name field must have been previously declared as a SETA symbol in a GBLA or LCLA instruction. The assembler evaluates the arithmetic expression in the operand field as a signed 32-bit arithmetic value and assigns this value to the SETA symbol in the name field. The SETA symbol in the name field can be subscripted, but only if the same SETA symbol is declared with an allowable dimension. If the symbol in the name field is subscripted, the assembler assigns the value of the expression in the operand field to the position in the declared array given by the value of the subscript. The subscript expression must not (1) be zero, (2) have a negative value, or (3) exceed the dimension you specified in the declaration.

```
* INSTRUCTIONS                          DESCRIPTION
*
          :
          LCLA  &A
          LCLA  &SUBA(90)               DECLARE 90-ELEMENT ARRAY
          :
&SUBA(20) SETA  2000                    SET 20TH ELEMENT=2000
          :
&A        SETA  &SUBA(20)               SET &A=20TH ELEMENT (=2000)
          :
&SUBA(99) SETA  1000                    **ERROR** ONLY 90 ELEMENTS
          :
```

SETC—Assign Character Value

SETC assigns a character string value to a SETC symbol. You can assign whole character strings or concatenate several smaller strings. The assembler assigns the composite string to your SETC symbol. A maximum of 64 characters is allowed in the composite character string. You can also assign parts of a character string to a SETC symbol by using substring notation.

You can change the character value assigned to a SETC symbol. This allows you to use the same SETC symbol with different values for character comparisons in several places or for substituting different values into the same model statement.

The format of the SETC instruction is:

Name	*Operation*	*Operand*
symbol	SETC	one of 4 options

The variable symbol in the name field must have been previously declared as a SETC symbol in GBLC or LCLC instruction. The four options you can specify in the operand field are:

- A type attribute reference
- A character expression
- A substring notation
- A concatenation of substring notations, character expressions or both

The assembler assigns the first 64 characters of the character string in the operand field to the SETC symbol in the name field.

Note. When you code a SETA or SETB symbol in a character expression, the unsigned decimal value of the symbol (with leading zeros removed) is the character value given to the symbol.

SETC Instruction	*Value of variable symbol*	*Value of SET symbol*
&C1 SETC T'&DATA	&DATA = RST	U
&C2 SETC 'ABC'		ABC
&C3 SETC 'ABCDE' (1,3)		ABC
&C4 SETC 'ABC'.'DEF'		ABCDEF
&C5 SETC '&A'	&A = 200	200
&C6 SETC '&A'	&A = 00200	200
&C7 SETC '&A'	&A = 200	200
&C8 SETC '-200'		-200
&C9 SETC '&A'	&A = 0	0
&C10 SETC '00200'		00200
&C11 SETC '&A+11'	&A = 30	30 + 11
&C12 SETC '1 - &A'	&A = - 30	1-30

The SETC symbol in the name field can be subscripted, but only if the same SETC symbol has been previously declared in a GBLC or LCLC instruction with an allowable dimension. If the symbol in the name field is subscripted, the assembler assigns the character value represented by the operand field to the position in the declared array given by the value of the subscript. The subscript expression must not (1) be zero, (2) have a negative value, or (3) exceed the dimension you specified in the declaration.

```
* INSTRUCTIONS                    DESCRIPTION
*
          :
          LCLC  &C
          LCLC  &SUBC(20)         DECLARE 20-ELEMENT ARRAY
          :
&SUBC(10) SETC  'ABC'             SETS TENTH ELEMENT=ABC
          :
&C        SETC  &SUBC(10)         SETS &C=TENTH ELEMENT(=ABC)
          :
&SUBC(25) SETC  'DEF'             **ERROR** ONLY 20 ELEMENTS
          :
```

SETB—Assign Binary Value

SETB assigns a binary bit value to a SETB symbol. You can assign bit values zero or one to a SETB symbol directly and use it as a switch. If you specify a logical expression in the operand field, the assembler evaluates this expression to determine whether it is true or false and then assigns the values one or zero, respectively. You can use the computed value in condition tests or for substitution.

The format of the SETB instruction is:

Name	*Operation*	*Operand*
variable symbol	SETB	one of 3 options

The symbol in the name field must have been previously declared as a SETB symbol in a GBLB or LCLB instruction. The 3 options you can specify in the operand field are:

- A binary value, 0 or 1
- A binary value enclosed in parentheses, (0) or (1)
- A logical expression enclosed in parentheses

The assembler evaluates a logical expression and determines if it is true or false. If it is true, it is given a value of 1; if it is false, a value of 0. The assembler assigns the explicitly specified binary value (0 or 1) or the computed logical value (0 or 1) to the SETB symbol in the name field. For example:

```
* SETB INSTRUCTION                VALUE ASSIGNED
*
&B1       SETB  0                  0
&B2       SETB  (1)                1
&B3       SETB  (2 GT 3)           0
&B4       SETB  (2 LT 3)           1
```

The SETB symbol in the name field can be subscripted, but only if the same SETB symbol has been previously declared in a GBLB or LCLB instruction with an allowable dimension. If the symbol in the name field is subscripted, the assembler assigns the binary value explicitly specified or implicit in the logical expression to the position in the declared array given by the value of the subscript. The subscript expression must not (1) be zero, (2), have a negative value, or (3) exceed the dimension specified in the declaration. For example:

```
* INSTRUCTIONS                    DESCRIPTION
*
          :
          LCLB   &B
          LCLB   &SUBSCR(50)      DECLARES 50-ELEMENT ARRAY
          :
&SUBSCR(10) SETB 1                SETS TENTH ELEMENT=1
          :
&B        SETB   &SUBSCR(10)      SETS &B=TENTH ELEMENT(=1)
          :
&SUBSCR(72) SETB 1                **ERROR** ONLY 50 ELEMENTS
          :
```

Using Expressions in SET Instructions

You can use three types of expressions in conditional assembly instructions: arithmetic, character, and logical. The assembler evaluates these conditional assembly expressions at preassembly time.

Do not confuse the conditional assembly expressions with the absolute or relocatable expressions used in other assembler language instructions. The assembler evaluates absolute and relocatable expressions at assembly time.

Arithmetic (SETA) Expressions

You can use an arithmetic expression to assign an arithmetic value to a SETA symbol, or to provide subscripting values during conditional assembly processing.

An arithmetic expression can contain one or more SET symbols. This allows you to use arithmetic expressions wherever you wish to specify varying values, for example as:

- Subscripts for SET symbols
- Subscripts for symbolic parameters
- Subscripts for & SYSLIST
- Substring notation

Thus you can control loops, vary the results of computations, and produce different values for substitution into the same model statement.

Arithmetic expressions can be used as shown in the following table.

Can be used in	*Uses as*	*Example*
SETA instruction	operand	&A1 SETA &A1+2
AIF instruction or SETB instruction	comparand in arithmetic relation	AIF (&A*10 GT 30) .A
Subscripted SET symbols	subscript	&SETSYM(&A+10–&C)
Substring notation	subscript	'&STRING' (&A*2, &A – 1)
Sublist notation	subscript	Sublist: (A, B, C, D) When &A = 1 the value of &PARAM (&A + 1) is B.
&SYSLIST	subscript	&SYSLIST (&M + 1, &N – 2) &SYSLIST (N '&SYSLIST)
SETC instruction	character string in operand	&C SETC '5–10*&A' If &A = 10 then &C = 5 – 10*10.

Note. When an arithmetic expression is used in the operand field of a SETC instruction, the assembler assigns the character string representing the arithmetic expression to the SETC symbol, after substituting values into any variable symbols. It does not evaluate the arithmetic expression.

An arithmetic expression consists of one or more *arithmetic terms* combined with the arithmetic operators + (addition), – (subtraction), * (multiplication), and / (division). An arithmetic term can be any of the following:

- self-defining term
- count or number attribute reference
- variable symbol as follows

Variable symbols allowed as terms in an arithmetic expression are:

Variable symbol	*Restriction*	*Example*	*Value of example*
SETA	None	----	----
SETB	None	----	----
SETC	Value must be an unsigned decimal self-defining term	&C	123
Symbolic parameters	Value must be a self-defining term	&PARAM	X'A1'
&SYSLIST (n) or &SYSLIST (n, m)	Corresponding operand or sublist entry must be a self-defining term	&SYSLIST(3) &SYSLIST (3,2)	24 B'101'
&SYSNDX	None	----	----

Coding Conditional Assembly Arithmetic Expressions

The following is a summary of coding rules for arithmetic expressions:

- Only binary operators are allowed in arithmetic expressions.
- An arithmetic expression must not begin with an operator, and it must not contain two operators in succession.
- An arithmetic expression must not contain two terms in succession.
- An arithmetic expression must not have blanks between an operator and a term.
- An arithmetic expression can contain up to 16 terms and up to five levels of parentheses. The parentheses required for sublist, substring, and subscript notation count toward this limit.

Evaluation of Arithmetic Expressions

The assembler evaluates arithmetic expressions at preassembly time as follows.

- It evaluates each arithmetic term.
- It performs arithmetic operations from left to right; however, it performs the operations of multiplication and division before the operations of addition and subtraction.
- In division, it gives an integer result; any fractional portion is dropped. Division by zero gives a zero result.
- In parenthesized arithmetic expressions, the assembler evaluates the innermost expressions first and then considers them as arithmetic terms in the next outer level of expressions. It continues this process until the outermost expression is evaluated.
- The computed result, including intermediate values, must be in the range -2^{31} through $+2^{31}-1$.

For example, the expression

```
&A+(X'FF'*2+&B-(&C/2+K'&AREA))
```

would be evaluated in the order:

(1) evaluate & C
(2) evaluate & C/2
(3) evaluate K' & AREA
(4) evaluate result of (2) + result of (3)
(5) evaluate X'FF'*2
(6) evaluate & B
(7) evaluate result of (5) + result of (6)
(8) evaluate result of (7) − result of (4)
(9) evaluate & A
(10) evaluate result of (9)+ result of (8)

Note. Self-defining terms are limited by assembly-time constraints; they must be in the range -2^{16} through $+2^{16}-1$.

The performance time of the assembler may be affected by the way SETA expressions are coded if large values are being used. The timing on multiply operations will be improved if the larger of the two values is placed first in the expression. When possible, write the expression so that the partial results of division operations will be small values. For example, the expression 600/300*10 gives the same result as 10*600/300, but the first expression will be evaluated in less time than the second. However, loss of precision must be considered when using divide because fractional integers, in partial results, are dropped. For example: 2*9/6 gives a result of 3, while 9/6*2 gives a result of 2.

Arithmetic expression	*Value of variable symbol*	*Value of expression*
&A + 10/&B	&A = 10; &B = 2	15
(&A + 10) /&B	&A = 10; &B = 2	10
&A/2	&A = 10	5
&A/2	&A = 11	5
&A/2	&A = 1	0
10*&A/2	&A = 1	5

Character (SETC) Expressions

The main purpose of a character expression is to assign a character value to a SETC symbol. You can then use the SETC symbol to substitute the character string into a model statement. You can also use a character expression as a value for comparison in condition tests and logical expressions. In addition, a character expression provides the string from which you can select characters with substring notation.

Substitution of one or more character values into a character expression allows you to use the character expression wherever you need to vary values for substitution or to control loops.

You can use character (SETC) expressions in conditional assembly instructions only as follows:

- In SETC instruction as an operand; for example:
  & C SETC 'STRING0'
- In AIF or SETB instructions as a character string in character relation; for example:
  AIF ('& C' EQ 'STRING1').B
- In substring notation as the first part of the notation; for example:
  'SELECT'(2,5)
  where 'SELECT' is a character expression

A character expression consists of any combination of characters enclosed in apostrophes. Variable symbols are allowed. The assembler substitutes the representation of their values as character strings into the character expression before evaluating the expression.

Note. Up to 127 characters are allowed in a character expression. Attribute references are not allowed in character expressions.

Variable symbols used in character expressions are subject to the following restrictions:

Variable symbol	*Restrictions*	*Example*	*Value substituted*
SETA	Sign and leading zeros are suppressed; stand-alone zero is used	&A SETA 0–201 &C SETC '&A' &D SETC 0105 &ZERO SETA 0 &C SETC '&ZERO' &B SETB 1	–201 201 105 0 0 1
SETB	Must be 0 or 1	&B SETB 1 &C SETC '&B'	1 1
SETC	None	&C1 SETC 'ABC' &C2 SETC '&C1'	ABC ABC
Symbolic	None	& C1 SETC '&PARAM' if &PARAM is (ABC)	(ABC)
System variable symbols	None	&NUM SETC '&SYSNDX' if &SYSNDX = 0201 *Note.* Leading zeros are not suppressed	0201

Evaluation of Character Expressions

The value of a character expression is the character string within the enclosing apostrophes, after the assembler performs any substitution for variable symbols. Character expressions can be concatenated with each other or with substring notations in any order. You can then use the concatenated string in the operand field of a SETC instruction or as a value for comparison in a logical expression. The resultant string is the value of the expression used in conditional assembly operations: for example, the value assigned to a SETC symbol. Only the first 64 characters of the resultant string are assigned to a SETC symbol. You must code a double apostrophe to generate a single apostrophe as part of the value of a character expression. A double ampersand generates a double ampersand as part of the value of a character expression. To generate a single ampersand in a character expression, use the substring notation: for example ('& &'(1,1)). To generate a period following a variable symbol, either you must code 2 periods or the variable symbol must have a period as part of its value. You must code the concatenation character (a period) to separate the apostrophe that ends one character expression from the apostrophe that begins the next. For example:

Example	*Value of variable symbols used*	*Value of character expression*
'ABC'		ABC
'&PARAM'	&PARAM = SYMBOL	SYMBOL
'&A + 10'	&A = 10	10 + 10
'&A&A'	&A = 10	1010
'&C . &C'	&C = DEF	DEFDEF
'&C . ABC'	&C = DEF	DEFABC
'ABC&D'	&D = .	ABC.
'&E'	&E = null	null character string
'ABC&D. DEF'	&D = null	ABCDEF
'&C . . 505'	&C = 2	2.505
'&C.505'	&C = 2.	2.505
'ABC'. 'DEF'		ABCDEF
'ABC'.'ABCDEF' (4, 3)		ABCDEF
'&C' (4, 3). 'DEF'	&C = ABCDEF	DEFDEF
'&C' (4, 3) 'DEF'	&C = ABCDEF	DEFDEF
'ABC'. '&C'. 'DEF'	&C = null	ABCDEF
'ABC'. ' '. 'DEF'		ABCDEF

Logical (SETB) Expressions

You can use logical (Boolean) expressions to assign the binary value 1 or 0 to a SETB symbol. You can also use a logical expression to represent the condition test in an AIF instruction. This allows you to code a logical expression whose value (0 or 1) will vary according to the values substituted into the expression and thereby determine whether or not a preassembly branch is taken.

You can code logical (SETB) expressions in conditional assembly instructions only as follows:

- In SETB instructions as the operand; for example:

```
&B1       SETB  (&B2 OR 8 GT 3)
```

- In AIF instructions as the condition test part of the operand; for example:

```
AIF   (NOT &B1 OR 8 EQ 3).A
```

A logical expression consists of one or more logical terms connected by the logical operators:

- OR—addition
- AND—multiplication
- NOT—negation

The logical operators OR and AND must connect two logical terms; the logical operator NOT is a unary operator—it precedes a single logical term to indicate the negation of that term.

A *logical term* can be either:

- A SETB variable symbol, or
- A logical relation, which is one of the following:
  - An *arithmetic relation*: 2 arithmetic expressions separated by a relational operator
  - A *character relation*: 2 character strings (character expression, substring notation, type attribute reference, or concatenation of character expression and substring notation) separated by a relational operator

The relational operators are:

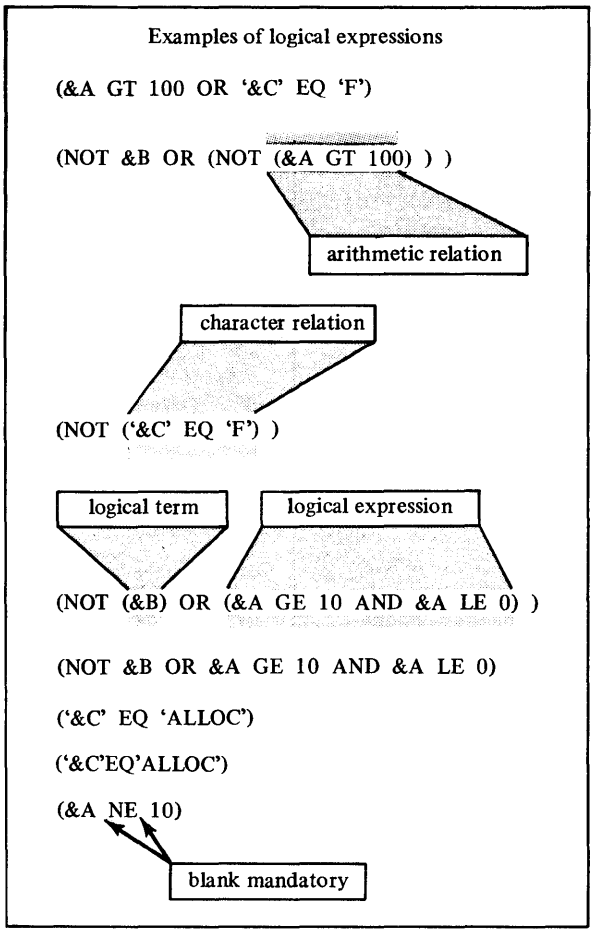
- EQ (equal)
- NE (not equal)
- LE (less than or equal)
- LT (less than)
- GE (greater than or equal)
- GT (greater than)

Rules for Coding Logical Expressions

A summary of coding rules for logical expressions follows:

- A logical expression must be enclosed in parentheses.
- A logical expression must not contain two logical terms in succession.
- A logical expression can begin with the logical operator NOT.
- A logical expression can contain two logical operators in succession; however, the only combinations allowed are: OR NOT or AND NOT. The two operators must be separated from each other by one or more blanks.
- Any logical term, relation, or inner logical expression can be optionally enclosed in parentheses.
- Relational and logical operators must be immediately preceded by a right parenthesis, a single quote, or at least one blank.
- Relational and logical operators must be immediately followed by an ampersand, a left parenthesis, a single quote, or at least one blank.
- A logical expression can contain up to 16 terms and up to five levels of parentheses.

Following are examples of logical expressions.



Evaluation of Logical Expressions

The assembler evaluates logical expressions as follows.

1. It evaluates each logical term, and assigns a binary value of 0 or 1.
2. If the logical term is an arithmetic or character relation, the assembler evaluates:
   a. The arithmetic or character expression specified as values for comparison, and then
   b. The arithmetic or character relations, and finally
   c. The logical term, which is the result of the relation. If the relation is true, the logical term it represents is given a value of 1; if the relation is false, the term is given the value of 0.

   Note. If two comparands in a character relation have character values of unequal length, the assembler always takes the shorter character value to be less than the longer one.
3. The assembler performs logical operations from left to right. However:
   a. It performs logical NOTs before logical ANDs and ORs, and
   b. It performs logical ANDs before logical ORs.
4. In parenthesized logical expressions, the assembler evaluates the innermost expressions first and then considers them as logical terms in the next outer level of expressions. It continues this process until the outermost expression is evaluated. For example, the expression

   ```
   ( NOT ( &B1 OR ( &B2 AND ( '&C' EQ 'X' OR &B3 ) ) ) )
   ```

   would be evaluated in the order:
   (1) evaluate '& C' EQ 'X'
   (2) evaluate the result of (1) OR & B3
   (3) evaluate & B2 AND the result of (2)
   (4) evaluate & B1 OR the result of (3)
   (5) evaluate NOT the result of (4)

Following are examples of logical expressions:

Examples of logical expressions

((&A NE 100) OR T'&AREA EQ '&PARAM'(3,4))

('ABC' LT 'ABCD') (Always true)

(&B AND NOT (5 GT 3))

equivalent

(&B AND (NOT (5 GT 3)))

(&B OR &A AND ('&C EQ 'B'))

equivalent

(&B OR (&A AND ('&C' EQ 'B'))

Selecting Characters From a String—Substring Notation

The substring notation allows you to refer to one or more characters within a character string. You can select characters from the string and use them for substitution or testing. By concatenating substrings with other substrings or character strings, you can rearrange and build your own strings.

Substring notation can be used only in the following conditional assembly instructions:

- SETC instruction as an operand or part of an operand; for example:
  - & C1 SETC 'ABC'(1,3)
    assigns the value ABC to & C1
  - & C2 SETC ' & C1'(1,2).'DEF'
    assigns the value ABDEF to & C2, based on & C1=ABC
- SETB and AIF instructions as a character value in the comparand of a character relation comprising part of a logical expression; for example:
  - AIF (' & STRING'(1,4) EQ 'AREA').SEQ
  - & B SETB (' & STRING'(1,4).'9' EQ 'FULL9')

Substring notation has the following format:

'character string'(e1,e2)

where the character string must be a valid character expression with a length N in the range 1–127 characters, and the subscripts e1 and e2 are arithmetic expressions. The first subscript, e1, indicates the first character that is extracted from the character string; the second subscript, e2, indicates the number of characters extracted (or the length of the substring). Substring notation is replaced by a value that depends on the 3 elements N, e1, and e2 as follows:

- When e1 has a value of zero or a negative value, the assembler issues an error message.
- When the value of e1 exceeds N, the assembler issues a warning message, and a null character string is generated.
- When e2 has a value of zero, the assembler generates a null character string. Note that if e2 is negative, the assembler issues an error message.
- When e2 indexes past the end of the character expression (that is, e1+e2 is greater than N+1), the assembler issues a warning message and generates a substring that includes only the characters up to the end of the character expression (e2 must be less than or equal to 64).

The following examples indicate the results of valid and invalid substring notation:

- 'ABCDEF'(2,5)
  Valid; results in character value of BCDEF
- 'ABCDEF'(0,5)
  Invalid because e1=0; results in null character value
- 'ABCDEF'(7,5)
  Invalid because e1 is greater than N; results in null character value
- 'ABCDEF'(3,0)
  Invalid because e2=0; results in null character value
- 'ABCDEF'(3,5)
  Valid, but produces a warning message because e2 indexes past end of string; results in character value of CDEF (only 4 characters long)
- 'ABCDEF'(3,4)
  Valid; results in character value of CDEF

Branching

There are four conditional assembly instructions that control the sequence of execution of statements within a macro definition:

- AIF—Conditional branch
- AGO—Unconditional branch
- ACTR—Loop control counter
- ANOP—No Operation

AIF—Conditional Branch

AIF is used to:

- Branch according to the result of a condition test
- Provide loop control for conditional assembly processing
- Check for error conditions and branch to an appropriate MNOTE instruction

Code the AIF instruction as follows:

Name	*Operation*	*Operand*
sequence symbol or blank	AIF	logical expression enclosed in parentheses, immediately followed by a sequence symbol with no intervening blanks

The assembler evaluates the logical expression in the AIF operand field at preassembly time. If the logical expression is true (logical value=1), the next statement processed by the assembler is the statement identified by the sequence symbol in the operand field of the AIF instruction; if the logical expression is false (logical value=0), the next sequential statement is processed next. The sequence symbol in the operand field is a conditional assembly label (the name field of a model statement or another conditional assembly instruction) that represents a location at preassembly time; the label can appear before or after the AIF instruction, within the same macro definition as the corresponding AIF instruction.

The following example indicates the use of the AIF instruction:

```
         MACRO
         MACAIF
         :
.BACK    AIF   ('&C' EQ 'F').FORWARD
         :
         AIF   (&A GT 5).BACK
.FORWARD ANOP
         :
         MEND
```

AGO—Unconditional Branch

AGO is used to branch unconditionally. This provides you with final exits from conditional assembly loops.

Code the AGO instruction as follows:

Name	*Operation*	*Operand*
sequence symbol or blank	AGO	sequence symbol

The statement identified by the sequence symbol in the AGO instruction operand can appear before or after the AGO instruction, within the same macro definition as the corresponding AGO instruction.

The following example indicates the use of the AGO instruction:

```
         MACRO
         MACAGO
         :
         AGO    .FORWARD
         :
.BACK    ANOP
         :
         AGO    .BACK
         :
.FORWARD ANOP
         :
         MEND
```

ACTR—Assembly Loop Counter

ACTR is used to set a conditional assembly loop counter. Each time the assembler processes an AIF or AGO branching instruction, the loop counter for that macro definition is decreased by one. When the number of conditional assembly branches taken reaches the value assigned by the ACTR instruction, the assembler exits from the macro definition.

By using the ACTR instruction, you avoid excessive looping during conditional assembly processing at preassembly time (in case of errors).

The format of the ACTR instruction statement is:

Name	*Operation*	*Operand*
blank	ACTR	any valid SETA expression

The ACTR instruction, if used, must be the first statement following global and local declarations for the macro definition.

A conditional assembly loop counter is set to the value of the arithmetic expression in the operand field. The loop counter has a local scope; its value is decreased only by AGO and AIF instructions (if the branch is taken). The loop counter is reset each time the macro is called. The nesting of macros has no effect on the setting of individual loop counters.

The assembler sets its own internal loop counter for each macro definition that does not contain an ACTR instruction. The assembler assigns a standard value of 150 to each of these internal loop counters.

Within the local scope of a particular loop counter (including the internal counters run by the assembler), the following rules apply.

- Each time the assembler executes an AGO or AIF branch, it checks the loop counter for a zero value.
- If the count is not zero, it is decreased by one.
- If the count is zero, before decreasing the counter value, the assembler terminates the expansion of the entire nest of macro definitions and processes the next sequential instruction after the outer macro call.

ANOP—Assembly No Operation

You can specify a sequence symbol in the name field of an ANOP instruction, and use the symbol as a label for branching purposes. The ANOP instruction performs no operation itself. Instead, if you branch to an ANOP instruction, the assembler processes the next sequential instruction. You use it preceding an instruction that already has a symbol in its name field. For example, if you wanted to branch to a SETC instruction, which requires a variable symbol in the name field, you would insert a labeled ANOP instruction immediately before the SETC instruction. By branching to the ANOP instruction with an AIF or AGO instruction, you would, in effect, be branching to the SETC instruction.

The format of the ANOP instruction statement is:

Name	*Operation*	*Operand*
sequence symbol	ANOP	blank

For example:

```
          :
          AGO   .SEQ
          :
.SEQ      ANOP
&A        SETA  1Ø
          :
```

Chapter 7. Using the Macro Assembler

Section Contents

Assembler Options

You may specify assembler options to the assembler through the JSP PARM control statement. The following list explains each of the options–the defaults are underlined.

Option	*Explanation*
LIST/NOLIST	LIST tells the assembler to write all assembly listings to the PRINT file. The listings are (1) external symbol dictionary (ESD), (2) source and object programs, (3) relocation dictionary (RLD), (4) cross-reference (XREF) and (5) diagnostic messages. NOLIST tells the assembler to write only error messages to the PRINT file.
TEXT/NOTEXT	TEXT tells the assembler to write the source and object program listing to the PRINT file. NOTEXT suppresses this processing.
XREF/NOXREF/FULLXREF	XREF tells the assembler to write the cross-reference listing to the PRINT file for only referenced symbols. NOXREF suppresses this processing. FULLXREF tells the assembler to write the cross-reference listing to the PRINT file for all defined and referenced symbols.
ESD/NOESD	ESD tells the assembler to write the external symbol dictionary before the source program listing. NOESD suppresses this processing.
RLD/NORLD	RLD tells the assembler to write the relocation dictionary after the source program listing. NORLD suppresses this processing.
SYSPARM('...')	Defines up to 8 characters of information substituted for the &SYSPARM value during macro processing. Blanks may be contained within the apostrophes. Each blank counts as one character. A single apostrophe within the string must be represented by two apostrophes. The default value for &SYSPARM is a null character string.
OBJECT/NOOBJECT	OBJECT causes the object module to be written to the OBJOUT file. NOOBJECT suppresses this processing.
LINECOUNT (n)	LINECOUNT specifies n as the number of lines per page of the PRINT file. If this option is omitted, the default line count is 55. The value of n must be in the range 1–999.
MACRO/NOMACRO	MACRO tells the assembler to process any macros encountered in the source. NOMACRO tells the assembler that there are no macros in the source and that macro processing may be bypassed.

Options are processed in the order they are specified. For example, if you enter

```
NOLIST,ESD
```

your output will be an ESD listing and diagnostic messages. NOLIST turns off the ESD, source, RLD, and XREF listings. ESD turns the ESD listing option back on.

Required Files

- Source program input file, SOURCIN
- Three assembler workfiles, WORK1, WORK2, WORK3, and/or WORKVOL.
- Program listing output file, PRINT.
- Assembler phase and overlay file, TSOVLY.

Optional Files

- Libraries, LIB1, LIB2. The libraries contain copy code, system macros, and user provided macros. LIB1 is searched before LIB2.
- Object module output file, OBJOUT

Assembler Program Listing

The assembler listing consists of six sections ordered as follows:

- External symbol dictionary
- Source and object program
- Relocation dictionary
- Symbol cross-reference table
- Diagnostic messages
- Statistics

The contents of the listing are controlled by the assembler options list.

External Symbol Dictionary (ESD)

This section of the listing contains the external symbol dictionary information passed to the application builder in the object module. The entries describe the control sections, external references, and entry points in the source module. Six types of entries with their associated fields are shown in the following chart. The circled numbers refer to the corresponding heading in the sample program listing. The Xs indicate entries accompanying the designator for each type.

❶	❷	❸	❹	❺	❻
SYMBOL	*TYPE*	*ID*	*ADDR*	*LENGTH*	*LD ID*
X	SD	X	X	X	–
X	LD	–	X	–	X
X	ER	X	–	–	–
–	PC	X	X	X	–
X	WX	X	–	–	–
X	CM	X	–	X	–
X	GL	X	–	X	–
X	RR	X	–	–	–

1. This column contains the name of every control section, entry point, and external symbol.
2. This column contains the type designator for the entry, as shown in the table. The type designators are defined as:
   - SD *Section definition.* The symbol appears in the name field of a CSECT or START statement.
   - LD *Label definition.* The symbol appears as the operand of the ENTRY statement.
   - ER *External reference.* The symbol appears as the operand of the EXTRN statement or is defined as a V-type address constant, or an external branch instruction; for example, BALX.
   - PC *Private code.* Unnamed control section definition.

WX *Weak external reference.* The symbol appears as the operand of WXTRN statement, or is defined as a W-type address constant.

CM *Common section.* The symbol appears in the name field of a COM statement.

GL *Global section.* The symbol appears in the name field as a GLOBL statement.

RR *Resource reference.* The symbol is defined as an N-type constant.

3. This column contains the external symbol dictionary identification number (ESDID), a unique 4-digit hexadecimal number identifying the entry. It is also used in an LD entry and in the relocation dictionary for cross-referencing the ESD. The assembler assigns this number in sequence as the items are encountered in your source program.
4. This column contains the address of the symbol (hexadecimal notation) for SD and LD type entries, and is blank for ER, CM, GL, RR, and WX entries. For PC and SD entries, it indicates the starting address of the control section.
5. This column contains the assembled length, in bytes, of the control section (hexadecimal notation).
6. This column contains, for LD entries only, the ESDID assigned to the control section that defines the entry symbol.

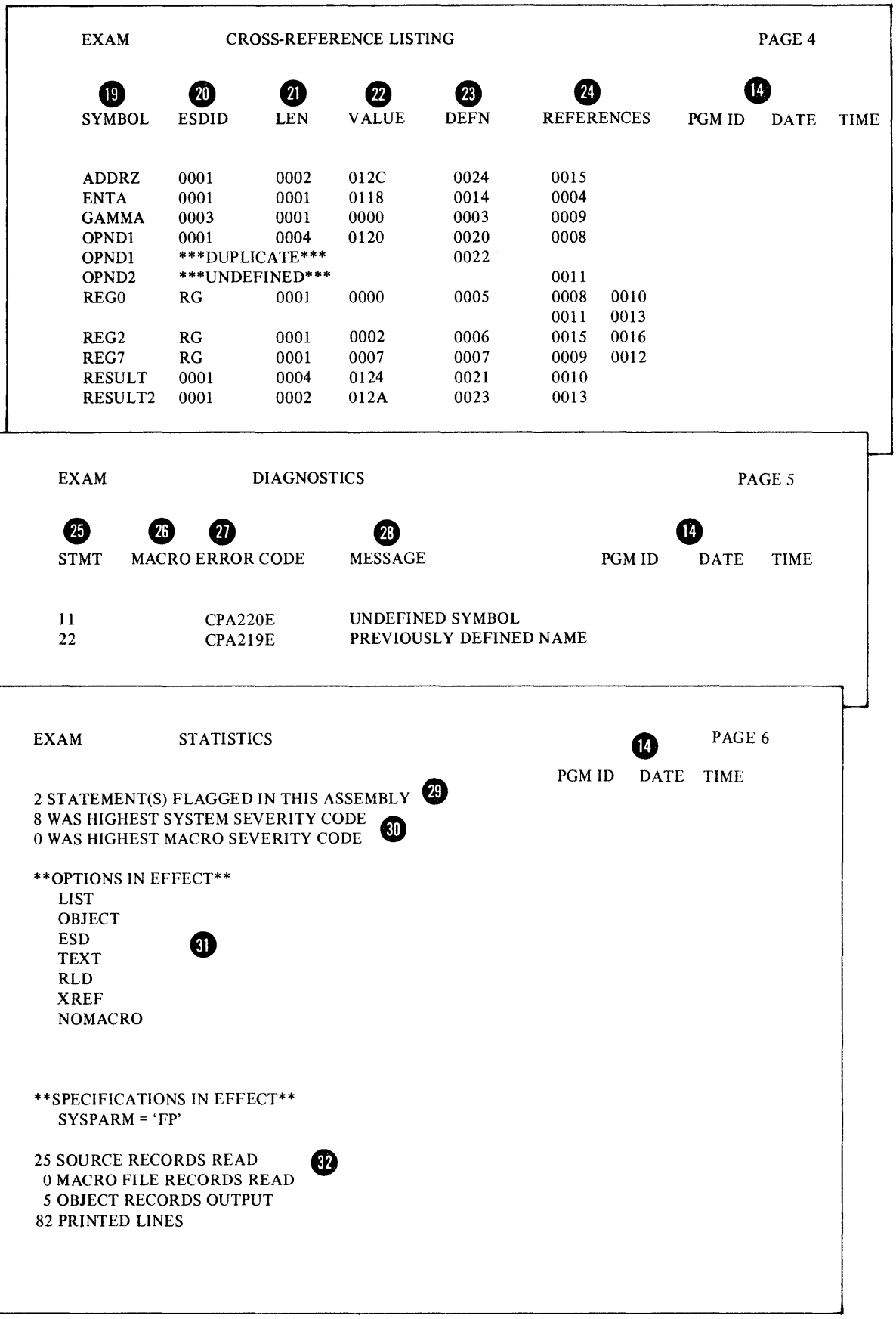
Source and Object Program

This section shows, on the next two pages, a sample assembly listing. An explanation of each part of the listing follows the sample.

```
EXAM          EXTERNAL SYMBOL DICTIONARY                    PAGE 1

 (1)          (2)   (3)     (4)      (5)       (6)      (14)
SYMBOL        TYPE  ID      ADDR     LENGTH    LD ID    PGM ID   DATE   TIME

SECTA         SD    0001    0100     002E
GAMMA         ER    0002
ENTA          LD            0118               0001
ALPHA         ER    0003
PROCZ         WX    0004
```

```
EXAM (7)                SAMPLE ASSEMBLY LISTING (8)          (9) PAGE 2

 (10)          (11)          (12)  (13)
LOC           OBJECT CODE   STMT  SOURCE  STATEMENT          (14) PGM ID  DATE   TIME

0100                        2     SECTA   START  X'0100'
                            3             EXTRN  GAMMA
                            4             ENTRY  ENTA
0000                        5     REG0    EQUR   0
0002                        6     REG2    EQUR   2
0007                        7     REG7    EQUR   7
0100          D020 0120     8             MVD    OPND1,REG0
0104          6F03 0000     9             BAL    GAMMA,REG7
0108          D028 0124     10            MVD    REG0,RESULT
010C          6808 0000     11            MVW    OPND2,REG0
***ERROR***
0110          6F03 0000     12            BALX   ALPHA,REG7
0114          680D 012A     13            MVW    REG0,RESULT2
0118                        14    ENTA    EQU    *
0118          6A08 012C     15            MVW    ADDRZ,REG2
011C          6842 0000     16            B      (REG2)
                            17    *
                            18    *          DATA AREA
                            19    *
0120          0000000A      20    OPND1   DC     D'10'
0124                        21    RESULT  DS     D
0128          001E          22    OPND1   DC     F'30'
***ERROR***
012A                        23    RESULT2 DS     F
012C          0000          24    ADDRZ   DC     W(PROCZ)
0000                        25            END
```

```
EXAM                    RELOCATION DICTIONARY                 PAGE 3

 (15)        (16)        (17)      (18)                        (14)
REL. ID      POS. ID     FLAGS     ADDRESS   REL. ID   POS. ID   PGM ID   DATE   TIME
                                                                FLAGS    ADDRESS

0001         0001        00        0102      0002      0001      10       0106
0001         0001        00        010A      0003      0001      10       0110
0001         0001        00        0116      0001      0001      00       011A
0004         0001        10        012C
```



EXAM CROSS-REFERENCE LISTING PAGE 4

⑲ SYMBOL	⑳ ESDID	㉑ LEN	㉒ VALUE	㉓ DEFN	㉔ REFERENCES	⑭ PGM ID	DATE	TIME
ADDRZ	0001	0002	012C	0024	0015			
ENTA	0001	0001	0118	0014	0004			
GAMMA	0003	0001	0000	0003	0009			
OPND1	0001	0004	0120	0020	0008			
OPND1	***DUPLICATE***			0022				
OPND2	***UNDEFINED***				0011			
REG0	RG	0001	0000	0005	0008	0010		
					0011	0013		
REG2	RG	0001	0002	0006	0015	0016		
REG7	RG	0001	0007	0007	0009	0012		
RESULT	0001	0004	0124	0021	0010			
RESULT2	0001	0002	012A	0023	0013			

EXAM DIAGNOSTICS PAGE 5

㉕ STMT	㉖ MACRO	㉗ ERROR CODE	㉘ MESSAGE	⑭ PGM ID	DATE	TIME
11		CPA220E	UNDEFINED SYMBOL			
22		CPA219E	PREVIOUSLY DEFINED NAME			

EXAM STATISTICS ⑭ PGM ID DATE TIME PAGE 6

2 STATEMENT(S) FLAGGED IN THIS ASSEMBLY ㉙
8 WAS HIGHEST SYSTEM SEVERITY CODE
0 WAS HIGHEST MACRO SEVERITY CODE ㉚

OPTIONS IN EFFECT
LIST
OBJECT
ESD ㉛
TEXT
RLD
XREF
NOMACRO

SPECIFICATIONS IN EFFECT
SYSPARM = 'FP'

25 SOURCE RECORDS READ ㉜
0 MACRO FILE RECORDS READ
5 OBJECT RECORDS OUTPUT
82 PRINTED LINES

7. This is the 4-character object module identification. It is the symbol that appears in the name field of the first TITLE statement. The assembler prints the TITLE statement identification and program identification (item 14) on every page of the listing.
8. This is the information taken from the operand field of a TITLE statement.

   Note. TITLE, SPACE, and EJECT statements do not appear in the source listing.
9. This is the listing page number.
10. This column contains the location counter value (hexadecimal notation) of the object code. For EQU instructions, this column contains the assembled value of the operand field.
11. This column contains the object code assembled from source statements. The entries are always left-justified. The notation is hexadecimal. Entries are either machine instructions or data constants. Machine instructions are printed in full with a blank inserted after every 4 digits (one word). Constants might be only partially printed, depending on the PRINT option in effect.
12. This column contains the statement number. A plus sign (+) to the right of the number indicates that the statement was generated as the result of expanding a macro instruction.
13. This column contains the source program statement. The following items apply to this section of the listing:
    - Source statements are listed, including macro definitions submitted in the source module.
    - Listing control instructions are not printed, with one exception. PRINT is listed when PRINT ON is in effect.
    - The statements generated as the result of a macro instruction follow the macro instruction in the listing unless PRINT NOGEN is in effect.
    - Diagnostic messages are not listed inline in the source and object program section. An error indicator, ***ERROR***, follows the statement in error, and appears inline when errors occur during macro definition expansion in NOGEN mode. (One or more of these indicators appear following the macro call, depending on the number of definition statements in error.) The message appears in the diagnostic section of the listing.
    - MNOTE messages are listed inline in the source and object program section. They are printed even if the NOGEN option is in effect. An MNOTE indicator appears in the diagnostic section of the listing for MNOTE statements other than MNOTE *. The MNOTE message format is severity code, followed by message text.
    - The MNOTE * form of the MNOTE statement results in an inline message only. An MNOTE indicator does not appear in the diagnostic section of the listing.
    - When an error is found in a source macro definition, it is treated the same as any other assembly error: the error indication appears after the statement in error, and a diagnostic is placed in the list of diagnostics. An error encountered during the expansion of a macro instruction is indicated at the point of error in the expansion and the associated diagnostic message is placed in the list of diagnostics. Errors occuring in a macro expansion (print NOGEN mode) are flagged inline with the macro call.
    - If the END statement contains an operand, the transfer address appears in the location column (LOC).
    - In the case of CSECT, START, COM, GLOBL, and DSECT statements, the location field contains the starting or resuming address of these control sections.

- In the case of EXTRN, WXTRN, and ENTRY instructions, the location field and object code field are blank.
- For a USING statement, the location field contains the value of the first operand.
- For ORG statements, the location Field contains the address value of the ORG operand.
- For an EQU or EQUR statement, the location field contains the value assigned to the symbol in the name field.
- Generated statements always print in standard statement format. Because of this, a generated statement can occupy two continuation lines on the listing, unlike source statements, which are restricted to one continuation line.

14. Program identification. The assembler supplies this information, which identifies the assembler program. The assembly date and the time the assembly started is also printed.

Relocation Dictionary

This section of the listing contains the relocation dictionary information passed to the application builder in the object module. Each line of the listing contains up to three relocation dictionary entries. The entries describe all address constants in the source module that are affected by relocation.

15. This column contains the ESDID number assigned to the ESD entry for the control section in which the referenced symbol is defined, or the ESDID number assigned to an ER item in the ESD.
16. This column contains the ESDID number assigned to the ESD entry that describes the control section in which the address constant is used as an operand.
17. The 2-digit hexadecimal number in this column is interpreted as follows:
    First digit. A 0 indicates that the entry describes an A-type address constant. A 1 indicates that entry describes a V-type address constant. A 4 indicates the entry describes a W-type address constant. A 5 indicates the entry describes an N-type address constant.
    Second digit. A 0 indicates that the relocation factor must be added to this item. A 2 indicates that the relocation factor must be subtracted.
18. This column contains the location counter value of the address constant in the source module.

Cross-reference

This section of the listing contains symbolic names used in the source module as well as certain information corresponding to the use of each symbolic name. If the FULLXREF option is specified, all symbolic names used are listed. Otherwise, only referenced symbolic names are listed.

19. This column contains the symbolic names in alphabetic order.
20. This column specifies the external symbol dictionary identifier (ESDID) in hexadecimal notation for the symbolic name. For register symbols, this field contains RG. Register symbols are absolute. An ESDID of X'0000' specifies that the symbol value is absolute. An ESDID other than X'0000' specifies that the symbol value is relocatable and is associated with that identifier.
21. This column states the length attribute (decimal notation) assigned to the symbol.
22. This column contains either the address the symbol represents, or a value to which the symbol is equated (hexadecimal notation).
23. This column contains the number of the statement in which the symbol is defined (decimal notation). Predefined register symbols will have statement number 0.

24. This column contains, from left to right, in ascending order, the numbers (decimal notation) of all statements in which the symbol appears in an operand.

Notes.

1. A PRINT OFF listing control instruction does not affect the printing of the cross-reference section of the listing.
2. In the case of an undefined symbol, the assembler fills columns 20, 21, 22, and 23 with the message:
   UNDEFINED
3. In the case of duplicate symbols, the assembler fills columns 20, 21, and 22 with the message:
   DUPLICATE
4. Symbols appearing in V or W-type address constants do not appear in the cross-reference listing.

Diagnostics

This section of the listing contains the diagnostic messages issued as a result of error conditions encountered in the program. For actual messages, see *IBM Series/1 Program Preparation Subsystem Messages and Codes.*

25. This column contains the number of the statement in error.
26. This column contains the name of a macro definition whenever certain errors associated with that macro definition are encountered.
27. This column contains the message identification–assembler identifier, message number, and severity.
28. This column contains the message text. In many cases, the assembler indicates the vicinity of the error with a near operand column pointer.

An MNOTE indicator of the form SEVERITY CODE xxx–MNOTE STATEMENT appears in the "Diagnostics" section if an MNOTE statement other than MNOTE * is issued by a macro instruction. xxx is the severity code associated with the statement flagged. The MNOTE statement itself is inline in the source and object program section of the listing. The operand field of an MNOTE * is printed as a comment but does not appear in the "Diagnostics" section of the listing.

Note. Editing errors in macro definitions from the macro source file are discovered when the macro definitions are read from the macro file. This occurs after the END statement has been read. They are therefore flagged after the END statement. The assembler lists the names of macro definitions along with error messages associated with those definitions. To help in isolating these types of editing errors, place the offending macro(s) into the source stream before the first control section.

Statistics

This section of the listing contains these statistical messages:

29. This is the number of statements flagged. The statements in error are printed in the "Diagnostics" listing.
30. This is the highest assembler severity code encountered, if not zero. Your macro severity code is also printed.

Code	*Message suffix*	*Meaning*
*		Informational message; no effect on execution
0	I	Informational message; normal execution is expected
4	W	Warning message; successful execution is probable
8	E	Error; execution may fail
12	S	Serious error; successful execution is improbable
20	T	Assembler program terminated abnormally

31. This is a list of the assembler options in effect.
32. This is the number of source records processed.

Performance

You may improve assembler performance by:

1. Increasing the size of the batch partition. This will allow the assembler to allocate larger I/O buffers and thereby, decrease the number of I/O operations.
2. Allocating SOURCIN and OBJOUT on the fixed disk. I/O is faster to the fixed disk than to the diskette.
3. Locating WORK1 on a second fixed disk. Leave WORK2, WORK3, SOURCIN and OBJOUT on the system disk. If a third disk is available, put WORK2 on that disk.
4. Not coding nested macros. Nested macros may cause additional I/O operations.

Invoking The Assembler (Example)

You must provide the assembler with the:

- Location of source input,
- Location of the assembler phases and overlays,
- Location of work volume or work files and
- Printer device.

An object file is required if you want object output. Libraries are required if you code *copy code* or want to access library macro definitions.

If you have set up the recommended program preparation default environment, with DSDs for assembler workfiles, object file, printer and system macro library, the following are the minimum JSP statements required to invoke the assembler. (Note that PPENV on the JOB statement is the name of a DSD statement describing a file on which the default DSDs reside):

```
ASM1      JOB   ENVL=PPENV
          EXEC  TSN=CPA
SOURCIN   DSD   DEV=MYDSKT,DSDTYPE=DISKETTE,VOL1=MYVOL,
                DSN=MYINPUT
          EOJ
```

WORK1, WORK2, and WORK3 default to the fixed disk specified in the default environment. The object file is created in the location specified in the default environment, and the default macro library that is specified is used. Source comes from data set 'MMYINPUT' on volume 'MYVOL', on device 'MYDSKT'. The printed output will go to the printer. Standard assembler default options will apply.

An example of an assembly with the source entered in the job stream is shown on page 7-10.

```
ASM2      JOB    ENVL=PPENV
          EXEC   TSN=CPA
SOURCIN   DSD    *
          MACRO
          AVG    &A,&B,&C          AVERAGE MACRO
          MVWS   &A,R6             ACCESS THE TWO
          MVWS   &B,R7             * PARAMETERS
          AW     R6,R7             ADD THEM TOGETHER
          SRL    ONE,R7            DIVIDE BY 2
          MVWS   R7,&C             SAVE AT SPECIFIED LOCATION
          MEND
PROGA     START  0
          PSW    R7,(R1)           SAVE LINK REGISTER
          AVG    (R0),(R2),(R3)    GET AVERAGE
          PW     (R1),R7           RESTORE R7
          BXS    (R7)              RETURN TO CALLER
ONE       EQU    1                 CONSTANT '1'
          END                      PROGA
          EOJ
```

All DSDs and assembler options default as in the first example, except for SOURCIN, which is spooled to a disk file by the job stream processor. The assembler will access that spool file.

The following is an example of JSP statements invoking the assembler with no environment list:

```
ASM1      JOB
          EXEC  TSN=CPA
ASMBLR    DSD   DEV=DSK1,DSDTYPE=DISK,VOL=ASMVOL,DSN=ASM
WORK1     DSD   DEV=DSK1,DSDTYPE=DISK,VOL=WRKVOL,DSN=WK1
WORK2     DSD   DEV=DSK1,DSDTYPE=DISK,VOL=WRKVOL,DSN=WK2
WORK3     DSD   DEV=DSK1,DSDTYPE=DISK,VOL=WRKVOL,DSN=WK3
SOURCIN   DSD   DEV=MYDSKT,DSDTYPE=DISKETTE,VOL1=MYVOL,
                DSN=MYINPUT
LIB1      DSD   DEV=DSK2,DSDTYPE=DISK,VOL=SYSLIB,DSN=MACLIB1
LIB2      DSD   DEV=DSK2,DSDTYPE=DISK,VOL=USERLIB,DSN=STRUCMAC
OBJOUT    DSD   DEV=DSK2,DSDTYPE=DISK,VOL=DECKOUT,DSN=MODULES,
                MEM=ASM11
PRINT     DSD   DEV=OPS,DSDTYPE=OPS
          PARM  'NOTEXT,NEXREF'
          EOJ
```

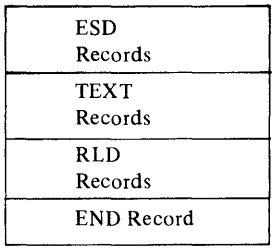
WORK1, WORK2, and WORK3 are in data sets 'WK1', 'WK2', and 'WK3' respectively, on volume 'WRKVOL', on device 'DSK1'. WORK1, WORK2, and WORK3 are accessed as direct type data sets with random organization. The source is accessed as in the previous example. Two macro libraries are on device 'DSK2': one in data set 'MACLIB1' on volume 'SYSLIB' and the other in data set 'STRUCMAC', on volume 'USERLIB'. The object output will go out to member 'ASM11' in data set 'MODULES', on volume 'DECKOUT', on device 'DSK2'. All printed output is routed to the 'OPS'. No assembly listing or cross reference listing will be output.

Object Module Formats

The macro assembler transforms source statements into object modules which are subsequently used as input to the application builder. The object module resides on a file which must be defined by a DSD statement named OBJOUT. This file can be a partitioned data set or consecutive data set.

Object modules are made up of ESD records, TEXT records, RLD records, and an END record. An object module always contains an ESD and an END record. Text is usually present in an object module; however, it is possible that none exist. The RLD is present only if there are relocatable address constants in the object module.

The following figure represents the general format of an object module contained in a consecutive data set or member of a partitioned data set on disk.



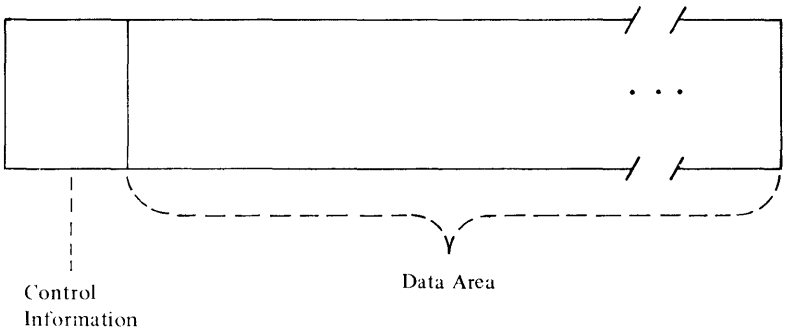
Object Module

Object module records are of variable length ranging from 16 to 512 bytes. Records are packed into a block of 128 or 256 bytes and may span blocks. The first record of an object module always begins a new block. The last block of an object module is written with the actual count of bytes present.

Record Formats

General Record Format

Object module records have two basic parts: control information and data area. The following figure represents the format:



Record Types

External Symbol Dictionary (ESD) Record

An ESD record contains one or more ESD entries where each entry is associated with a symbolic definition or reference within the object module text.

ESD Control Information

Offset	*Bytes*	*Field*	*Description*
0(0)	1	Code	Record code (X'02')
1(1)	3	ID	Record Identifier (C'ESD')
4(4)	2		Undefined
6(6)	2		Reserved
8(8)	2		Undefined
10(A)	2	Length	Number of bytes of ESD data
12(C)	2		Undefined
14(E)	2	ESDID	ESD entry identifier of the first entry for a symbol type other than LD.

ESD DATA

Offset	*Bytes*	*Field*	*Description*
0(0)	8	Symbol	Symbolic name in EBCDIC of the symbol described in this entry

8(8)	1	Type	A code which identifies the type of symbol being described. X'00'–Section definition (SD) X'01'–Label Definition (LD) X'02'–External Reference (ER) X'04'–Private Code (PC) X'05'–Common (CM) X'0A'–Weak External Reference (WX) X'0B'–Global (GL) X'0C'–Resource Reference (RR)
9(9)	1		Reserved
10(A)	2	Address	Address of the symbol within the object module when the type is SD, LD, or PC.
12(C)	2		Reserved
14(E)	2	ESDID/Length	The ESD entry of the control section containing the symbol if type is LD. For SD, PC, CM and GL type entries, this specifies the section length in bytes.

Text (TXT) Record

The instructions and data that make up a program are in text (TXT) records as text data. A given TXT record contains text data that is associated with a particular control section (CSECT).

TXT Control Information

Offset	*Bytes*	*Field*	*Description*
0(0)	1	Code	Record Code (X'02')
1(1)	3	ID	Record Identifier (C'TXT')
4(4)	2		Undefined
6(6)	2	Address	Address Assigned to the first byte of text data in this record relative to the beginning of the SD/PC entry that the text corresponds to.
8(8)	2		Undefined
10(A)	2	Length	Number of bytes of text data.
12(C)	2		Undefined
14(E)	2	ESDID	Identifier of the ESD entry for the control section (ESD type SD) to which the text is associated.

TXT Data

Offset	*Bytes*	*Field*	*Description*
0(0)	An even numbers	Data	Text data of a specified control section of a program.

Relocation Dictionary (RLD) Record

RLD records contain RLD entries. An RLD entry is produced for each address constant that is sensitive to the relocation of the program. RLD entries are generated for each relocatable A-type constant and for each V, W, and N-type constant within an object module.

RLD Control Information

Offset	*Bytes*	*Field*	*Description*
0(0)	1	Code	Record Code (X'02')
1(1)	3	ID	Record Identifier (C'RLD')
4(4)	2		Undefined
6(6)	2		Reserved
8(8)	2		Undefined
10(A)	2	Length	Number of bytes of RLD data.
12(C)	2		Undefined
14(E)	2		Reserved

RLD Data (per complete entry)

Offset	*Bytes*	*Field*	*Description*
0(0)	2	R-Pointer	Relocation Pointer, which is the ESDID number for the ESD entry describing the reference symbol.
2(2)	2	P-Pointer	Position Pointer, which is the ESDID number of the ESD entry for the control section referencing the symbol identified by the R-pointer.
4(4)	1	Flags	Indicates the direction of relocation, the type of relocatable constant, and the format of the next RLD data entry. B'TTTTUUDF' TTTT—Type of Constant 0000—A-type 0001—V-type 0100—W-type 0101—N-type UU—Undefined D—Direction of relocation 0—Positive; addition of the relocation factor. 1—Negative; subtraction of the relocation factor F—format of the next RLD data entry item. 0—Complete entry 1—Partial entry—R and P Pointers omitted (use same pointers as last complete RLD data entry)
5(5)	1		Undefined
6(6)	2	Address	Location (address) of the address constant requiring adjustment due to relocation.

RLD Data (per partial entry)

Offset	*Bytes*	*Field*	*Description*
0(0)	1	Flags	Indicates the direction of relocation, the type of relocatable constant, and the format of the next RLD data entry. See complete entry for format of the bit field.
1(1)	1		Undefined
2(2)	2	Address	Location (address) of the address constant requiring adjustment due to relocation.

End of Module (END) Record

There is one END record for an object module. It is the last record for that object module.

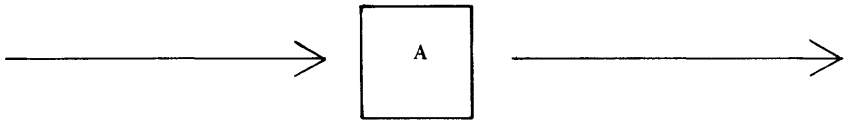
End Control Information

Offset	*Bytes*	*Field Description*	
0(0)	1	Code	Record Code (X'02')
1(1)	3	ID	Record Identifier (C'END')
4(4)	2		Undefined
6(6)	2	Address	Entry point address of the object module if specified in the END source statement.
8(8)	2		Undefined
10(A)	2	Length	Value of zero.
12(C)	2		Undefined
14(E)	2	ESDID	Identifier of the ESD entry for the control section containing the entry point specified in the address field.

Structured Programming Macros

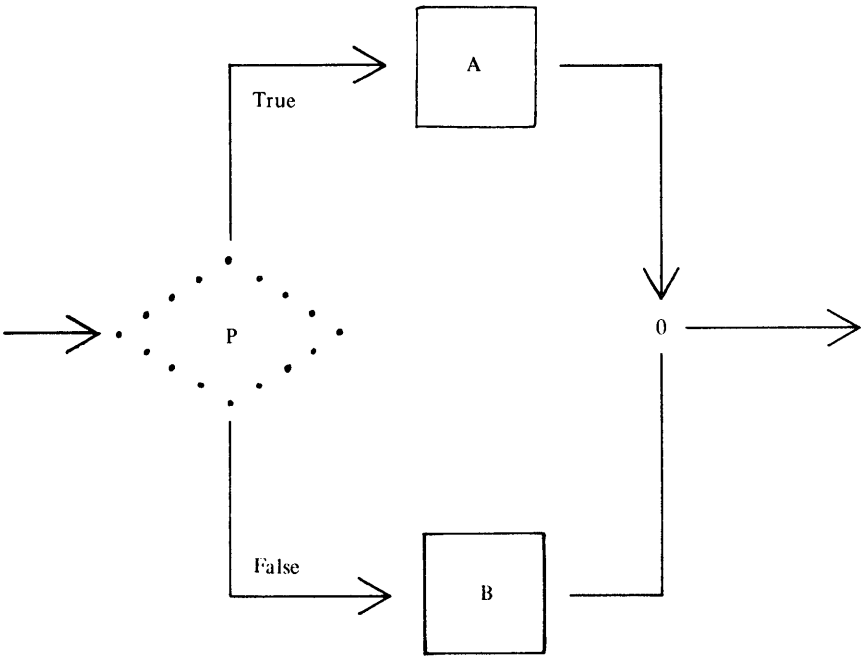
A set of structured programming macros (discussed in this section) are provided on a separate macro library which can be used as input to the assembler. This section describes the macros that support the following structures.

BLOCK



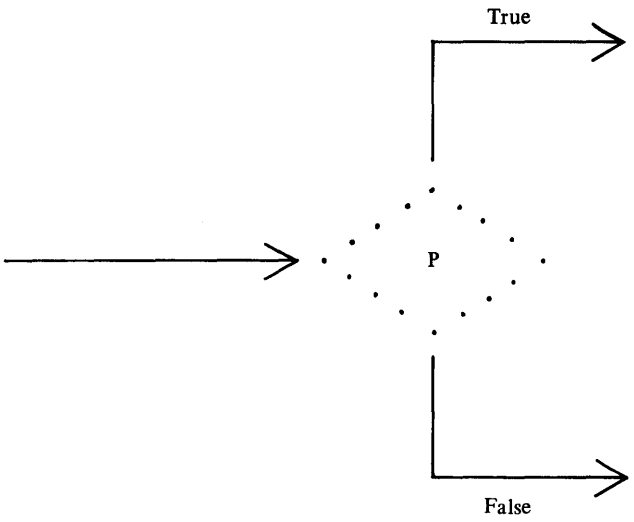
Note: All other structures must be nested within a block structure

IF THEN ELSE



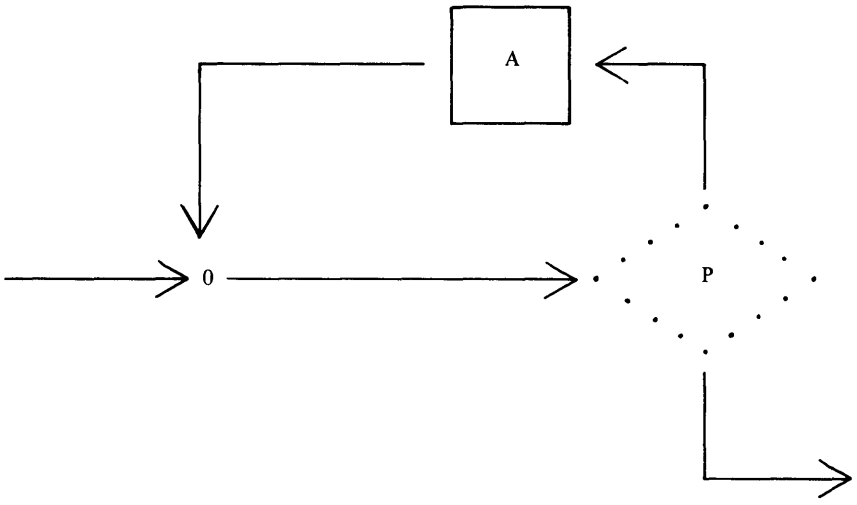
Note: Either A or B can be null

LEAVE

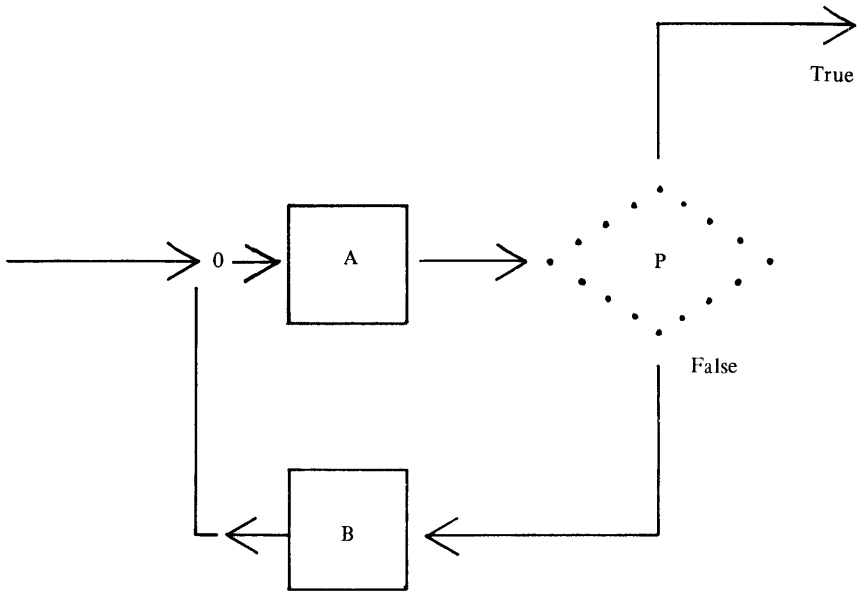


Note. LEAVE is not a proper structure. It is a compromise between code efficiency and the advantages of structured code (reduced development and maintenance cost, increased productivity, reduced errors, improved readability and understandability). See full discussion under macro syntax section.

DO WHILE

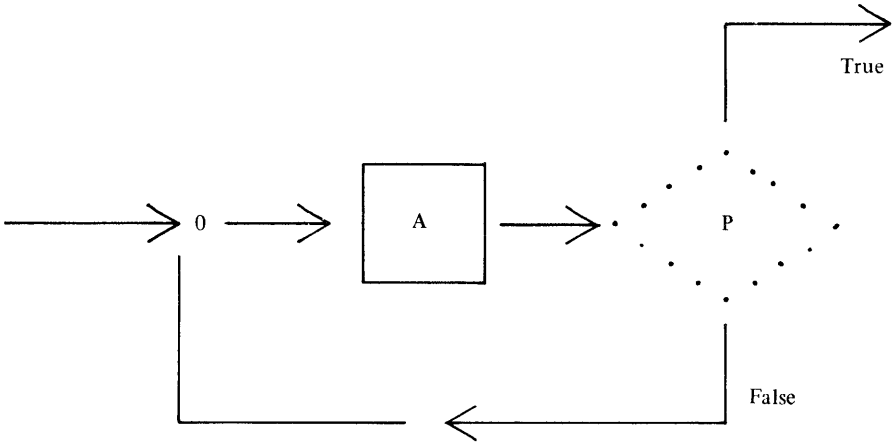


DO INFINITE

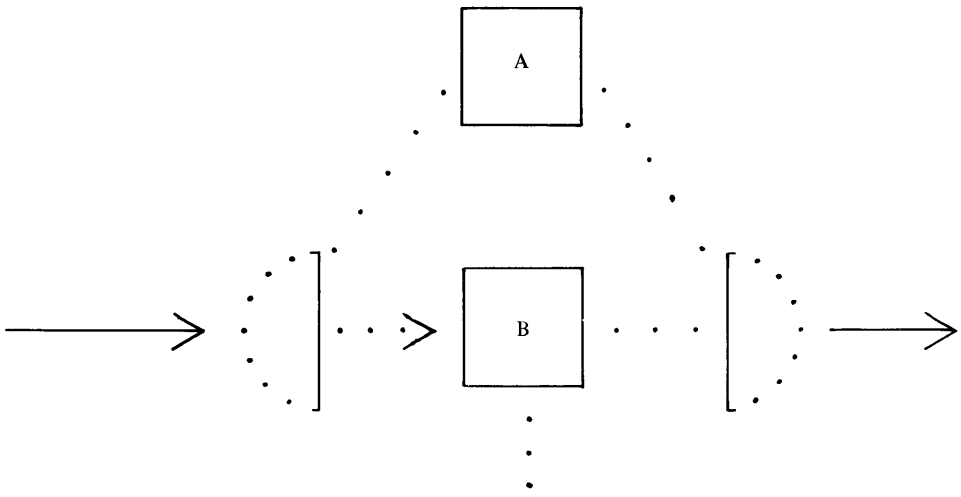


Note: Has LEAVE macro for exiting

DO UNTIL



CASE



Structured programming macros eliminate branch and jump instructions in source code. The code generated by the expansion of structured macros is reenterable.

To create these structures:	Use these macros:
BLOCK	BLOCK/ENDBLK
IF-THEN-ELSE	IF/[ELSE]/ENDIF or LEAVE
DO-WHILE, DO-INFINITE, and DO-UNTIL (depending on operands)	DO/ENDDO/LEAVE
CASE	CASEBLK/CASE/ENDCASE

Macro Syntaxes

BLOCK

A structured programming macro that indicates the beginning of a section of code to be treated as a unit.

```
[label] BLOCK [LEVELS={YES|NO}]
```

where:

label

Any unique label acceptable to the native assembler. This optional label is an aid to readability. Structured programming macros keep track of blocks by using internal nesting level numbers and therefore do not require labels on blocks.

If you code a label, your program can use it either as an operand on the LEAVE macro, telling it which structure you want to exit, or as an operand on the ENDBLK macro to verify that the two macros are related.

LEVELS={YES | NO}

The internal nesting level of every structured macro is available as a debugging aid through this optional keyword. When LEVELS=YES is coded, the nesting level of each structured programming macro appears as an MNOTE in each structured macro's expansion until a BLOCK macro with LEVELS=NO is encountered. If this keyword is never used then no nesting level MNOTEs are generated.

ENDBLK

A structured programming macro that indicates the end of a section of code to be treated as a unit.

```
[label] ENDBLK [BLOCK=blocklabel]
```

where:

label
Optional. Any unique label acceptable to the native assembler.

BLOCK=blocklabel
The label of the last preceding block that is not paired with an ENDBLK. Think of BLOCKS as left parentheses and ENDBLKS as right parentheses.

This optional keyword improves readability and allows verification. If the blocklabel provided does not match the label of the last preceding unpaired BLOCK, then an MNOTE noting the discrepancy is issued, and the last preceding unpaired BLOCK is ended.

IF

A structured programming macro that:

a. determines whether a statement or group of logically-connected statements are true or false, then
b. passes control to user-coded "true" code or "false" code.

```
[label] IF    statement[,{AND|OR},statement  X
              [,...]][,THEN][,END=FAR]
```

where:

label
Optional. Any unique label acceptable to the native assembler. If you code a label, your program can use it either as an operand on the LEAVE macro, telling it which structure you want to exit, or as an operand on the ENDIF macro to verify that the two macros are related.

statement
A statement to be tested by the IF macro. If the statement is true, control is given to the "true" code. If the statement is false, control is given to the "false" code. The "true" code is the code between the IF macro and the ELSE macro. The "false" code is the code between the ELSE macro and the ENDIF macro. If there is no "false" code, you can omit the ELSE macro, and the "true" code is the code between the IF macro and the ENDIF macro.

The statement to be tested can be one of two types:

1. The relationship between two items (for example, A is greater than B?)
2. The current status in the Level Status Register (for example, does the LSR indicate positive?)

A detailed presentation of all allowable statements follows this macro syntax discussion.

AND/OR
Statements can be logically connected. However, such logical statement strings are *not* evaluated according to normal Boolean reduction. They are evaluated this way:

- The macro evaluates the expression from left to right.
- If the next conjunction the macro encounters is OR, the macro checks whether the previous condition was true. If it was, the macro transfers control to the "true" code. If it was not true, the macro checks the next condition.
- If the next conjunction the macro encounters is *AND*, the macro checks whether the previous condition was false. If it was, the macro transfers control to the "false" code. If it was not false, the macro checks the next condition.
- If there are no more conjunctions, and the last statement is true, the macro passes control to the "true" code. Otherwise, it passes control to the "false" code.

THEN
An optional positional parameter that improves readability. This is also implemented as a macro itself. The THEN macro does nothing, generates no code, supports no labels, but is better documentation than the THEN positional parameter because the THEN macro may be aligned with the ELSE macro.

END=FAR
This optional keyword appears on every structured programming macro that implies a skip beyond the macro expansion itself. In the case of the IF macro, a skip from the IF around the "true" code to the "false" code is needed. By default the skip is generated in the form of a one-word jump instruction. If the user-provided code being skipped is very large, the range of a one-word jump might be exceeded. The END=FAR keyword tells the macro to generate a two-word branch instruction (range=64K bytes).

A suggested coding technique is to let all structured macros default to one-word jumps, and then code END=FAR on any flagged macro expansion.

THEN
A structured programming macro associated with IF macro and used solely to improve readability by allowing THEN to appear in the same column as the ELSE macro. THEN has no parameters and does not support labels.

ELSE

A structured programming macro associated with IF macro and used to separate "true" code from "false" code.

```
[label] ELSE [END=FAR]
```

where:

label
Optional. Any unique label acceptable to the native assembler.

END=FAR
See END keyword discussion in IF macro.

ENDIF

A sturctured programming macro associated with the IF macro. This macro indicates the end of user-written "false" code, if an ELSE macro is in effect, or the end of user-written "true" code, if an ELSE macro was not coded.

```
[label] ENDIF [IF=iflabel]
```

where:

label
Optional. Any unique label acceptable to the native assembler.

IF=iflabel
The label of the last preceding IF that is not already paired with an ENDIF. Think of the IFs as left parentheses and ENDIFs as right parentheses.

This optional keyword is provided for readability and verification. If the provided iflabel is not identical to the name of the last preceding unpaired IF, a warning MNOTE is generated, and the last preceding unpaired IF is ended.

LEAVE

A macro that passes control out of a named structure. Without this macro, you must set and test switches when leaving a nested structure. With LEAVE you can avoid this overhead, but you violate the structured properties of your code.

This macro has certain controls that keep its violations of structured principles to a minimum. These controls ensure that you are allowed to LEAVE only to the end of the current structure or to the end of an outer-nested structure. For

example, your program is executing code in structure C. Structure C is nested inside structure B, and structure B is nested inside structure A. You can LEAVE from C to the end of C, to the end of B, or to the end of A. In other words, you must be in the structure you are leaving.

You can use LEAVE in a way that preserves a proper structure. Using LEAVE to exit from a DO-INFINITE preserves structured coding as long as you LEAVE only the DO and not an outer-nested structure.

```
[label] LEAVE   [statement,]{BLOCK|IF|DO|CASEBLK}=
                structurelabel[,END=FAR]
```

where:

label
Optional. Any unique label acceptable to the native assembler.

statement
A statement that is one of the two types discussed under the IF macro. A full list of allowed statements follows this syntax section. If the statement you code is true, or if you do not code a statement, control leaves the specified structure. If the statement you code is false, LEAVE does not pass control out of the specified structure. The unconditional form of the LEAVE macro allows multiple entry points to modules, as approved in our programming specifications. If you use the unconditional LEAVE, be careful not to follow it with dead code.

Note. Unlike the IF and DO macros, the LEAVE macro does not permit logically-connected statement strings.

{BLOCK | IF | DO | CASEBLK}
Keyword for naming the highest-level structure in the set of containing structures that you wish to LEAVE.

Associating a structure name with a structure type improves readability and allows verification. If the structure you name is not of the type you specify, or if the LEAVE macro is not in the structure you name, an MNOTE is generated, and control passes to the end of the current structure.

END=FAR
See END= keyword discussion in IF macro.

DO

A structured programming macro that indicates the beginning of a user-coded program section through which the user wishes to loop.

```
[label]   DO    {WHILE|INF|UNTIL|CNTREG=reg}
          [,statement[,{AND|OR},statement[,...]]]
          [,END=FAR]
```

where:

label
Optional. Any unique label acceptable to the native assembler. If you code a label, your program can use it either as an operand on the LEAVE macro, telling it which structure you want to exit, or as an operand on the ENDDO macro to verify that the two macros are related.

WHILE | INF | UNTIL | CNTREG=reg
The first information you must specify to the DO macro is whether you wish to code a DO WHILE, DO INFINITE, or DO UNTIL. This is done by coding one of these three positional parameters, or the keyword CNTREG.

The keyword CNTREG generates a special form of DO UNTIL that is very efficient in both execution time and core utilization. Reg is a register (R0—R7) you have loaded with a value in the range 0—65535. This value is the number of times that the loop is executed. Because DO CNTREG is a trailing-decision loop, a count of either zero or one executes the DO once. The macro generates the

code to decrement the register. The drawbacks of the CNTREG form of the DO are as follows:

1. The UNTIL condition can only be the decrementing of the count to one.
2. You can have only 256 bytes of code between the DO CNTREG= macro and the ENDDO macro.

If you code DO INF, you must provide the mechanism for exiting from the loop. You can do this by coding the LEAVE macro (explained later), or, if you are executing an infinite GET I/O loop, by coding an END OF FILE exit address that is outside the loop.

Tests for exiting the remaining two forms of the DO macro (DO WHILE and the general form of the DO UNTIL) are provided through the following (statement) parameter.

statement
Valid only following WHILE or UNTIL, this is the loop exit test. A DO WHILE loop continues as long as this statement is true (a DO WHILE is a leading-decision loop). A DO UNTIL loop continues until this statement is true, at which time the loop is exited (a DO UNTIL is a trailing-decision loop). There are two types of statements, and they are introduced under the IF macro. A full list of allowed statements follows this syntax section.

AND/OR
Valid only following WHILE or UNTIL. Statements can be logically connected as discussed under the IF macro. The statement string nets out to true or false, as does a single statement. The netted-out string is handled as a single statement by DO WHILE and DO UNTIL. See preceding paragraph.

END=FAR
Not valid in DO CNTREG form. See END keyword discussion in IF macro.

ENDDO

A structured programming macro associated with the DO macro. ENDDO indicates the end of the DO loop.

```
[label] ENDDO [DO=dolabel]
```

where:

label
Optional. Any unique label acceptable to the native assembler.

DO=dolabel
The label of the last preceding DO that is not already paired with an ENDDO. Think of the DOs as left parentheses and ENDDOs as right parentheses.

This optional keyword improves readability and allows verification. If the dolabel you code is not the name of the last preceding unpaird DO, a warning MNOTE is generated, and the last preceding unpaired DO is ended.

CASEBLK

A structured programming macro that indicates that beginning (fan out) of a case structure. Coding the case structure is more efficient than coding multiple IFs. However, the CASE structure has the following limitation: A CASE structure cannot be nested within a CASE structure.

```
[label] CASEBLK   REG=register[,TREG=register][,END=FAR]
```

where:

label
Optional. Any unique label acceptable to the native assembler. If you code a label, your program can use it either as an operand on the LEAVE macro, telling

it which structure you want to exit, or as an operand on the ENDCASE macro to verify that the CASEBLK and ENDCASE macros are related.

REG=
A general register (R1—R7) that contains the current case value (0—254) upon entry to the CASEBLK macro. The number in this register determines which case (see CASE macro below) gets control on this pass through the case structure. The register need be set up only at entry to CASEBLK, that is, the register is available for any other use before and after the CASEBLK macro. CASEBLK alters the value it finds in the specified register.

TREG=
A general register (R0—R7). Must be a different register than the one selected for REG=. Coding this optional keyword produces a *less* efficient form of the CASEBLK/ENDCASE structure that has an additional restriction: the entire CASEBLK/ENDCASE structure may not span more than 256 bytes. Note that this restriction is encountered *only* when TREG= is coded. The single advantage of this form is that TREG= produces a CASEBLK/ENDCASE structure with no relocatable items in it. This form of the structure is the only form that runs when it is not possible to resolve RLDs. For example, transient code must be quickly read into one of several transient areas. There must be no RLDs in such code because time is not avaialable to resolve them as they are loaded, and because there are several equally possible transient areas available, the RLDs cannot be resolved before they are loaded.

END=
For use with TREG form of the structure. END=FAR removes the restriction that ENDCASE follow within 256 bytes of CASEBLK at the cost of doubling the size of the branch table generated by ENDCASE.

CASE

A structured programming macro that indicates the beginning of a section of user-provided code. The end of this CASE code is indicated either by another CASE macro or by an ENDCASE macro. This code is executed if the number associated with this CASE is in the CASEBLK register at entry to the CASEBLK macro.

```
[label] CASE n[,n...][,END=FAR]
```

where:

label
Optional. Any unique label acceptable to the native assembler.

n[,n...)
The number or numbers associated with this case. If any one of the numbers provided here is in the register coded on the CASEBLK macro, then CASEBLK passes control to this case. The number(s) must be unique; that is, once a number is assigned to a case, no other case within that entire case structure can use that number.

Note. Storage is conserved by using contiguous numbers beginning with 0. Thus, if you have five cases, use numbers 0 through 4, not 10, 20, 30, 40, and 50 or 201 through 205.

ENDCASE

A structured programming macro that indicates the end (fan in) of a case structure.

```
[label] ENDCASE   [CASEBLK=caseblklabel]
```

where:

label

Optional. Any unique label acceptable to the native assembler.

CASEBLK=
The label of the last preceding CASEBLK. This optional parameter improves readability and allows verification. If the caseblklabel you code is not the name of the last preceding CASEBLK, the CASE structure is closed and a warning MNOTE is issued.

Statements

A Statement tells the branching macros (IF, LEAVE, DO) under what conditions a skip is to be taken. The Level Status Register (LSR) is what all skip (Branch/Jump) instructions key on.

The LSR looks like this:

BIT	
0	EVEN Indicator
1	CARRY Indicator
2	OVERFLOW Indicator
3	NEGATIVE RESULT Indicator
4	ZERO RESULT Indicator
5—15	Other LSR Bits

A Statement always indicates which bits in the LSR are to be tested. Some Statements set the LSR bits before indicating which bits are to control the skip. Other statements just indicate which LSR bits are important and assume that you, in your preceding code, have set the LSR bits the way you want them. Said another way, some statements generate a compare and a skip, while other statements just generate a skip, assuming that you have coded the compare yourself ahead of the macro.

Conditional statements assume the LSR is already set up by you and just generate a skip. Conditional Statements are mnemonics that generate the proper LSR testing mask for the skip. A list of all supported conditional statements follows:

EQ	NE	equal, not equal
P	NP	positive, not positive
N	NN	negative, not negative
EV	NEV	even, not even
Z	NZ	zero, not zero
CY	NCY	carry, not carry
OV	NOV	overflow, not overflow
ON	NON	bit(s) on, bit(s) not on
OFF	NOFF	bit(s) off, bit(s) not off
MIX	NMIX	bits mixed, bits not mixed
LGT	LLE	logically greater than, less than or equal
LLT	LGE	logically less than, greater than or equal
GT	LE	arithmetically greater than, less than or equal
LT	GE	arithmetically less than, greater than or equal
CCn	NCCn	condition code 'n', not condition code 'n'
		where: n = an expression whose value is 0—7
ER	NER	error (NCC7), no error (CC7)

Relational statements do more than just generate a skip. They generate a compare and a skip. Because of the additional information required to generate the compare that alters the LSR, a relational statement is more complex than a conditional statement:

Relational Syntax

```
(p1,relcond,p2[,width])
or
(p1,IS,{ON|OFF|NON|NOFF}[,width])
```

where:

p1,p2
Data elements, coded with same syntax as assembler language operands because the macro merely places P1 and P2 as the first and second operands of the appropriate compare instruction without doing any syntax checking. If P2 is coded as an immediate, then the macro strips off the equal sign and sees that P2 is placed as the first operand of the compare immediate. Other than this minor exception no operations are performed on the data you supply as P1 and P2. So if you code improper syntax, the macro will not detect it, but the assembler will. See the section on 'Conditional—Setting Instructions' below for examples of assembler language syntax.

Note. If you wish the macro to generate a compare immediate, then code the immediate self-defining value or expression in the P2 position, preceded by an equal sign.

BYTE IMMEDIATE FORMS
=X'..', =B'.....', =C'.', =n, =expression, =H'n', =S'n'

WORD IMMEDIATE FORMS
=X'....',=B'.....',=C'..', =n, =expression
=F'n', =S'..'
IS,{ON | OFF} single-bit-testing form, in this syntax 'p1'
{NON | NOFF} must be (reg, bit disp)

relcond
A relational condition that indicates the relationship to be applied against the two data elements.

RELATIONAL MNEMONICS

Logical		
EQ	NE	equal, not equal
LGT	LLT	greater than, less than
LGE	LLE	greater than or equal, less than or equal
Arithmetic		
GT	LT	greater than, less than
GE	LE	greater than or equal, less than or equal
Bit Mask		
ON	NON	bit(s) on, bit(s) not on
OFF	NOFF	bit(s) off, bit(s) not off
MIX	NMIX	bits not same, bits same

width
Implied by one of the three preceding operands. Can always be coded if desired to improve readability.

WIDTH MNEMONICS

{BIT|BYTE|WORD|DWORD|FLOAT|DFLOAT|ADDR}

default=WORD

Relational Examples

((R2),LGT,=AB-B)	defaults to WORD
((R3,3),IS,ON)	implies BIT, via 'IS,ON'
(R1,EQ,=X'17')	implies BYTE, hexadecimal length
((R1),EQ,=C'/*')	implies WORD, string length
(R2,EQ,=C', ')	implies BYTE, string length
(A,EQ,B,BYTE)	specifies BYTE
(R4,EQ,=B'01011101')	implies BYTE, binary length
(A,ON,=B'1111100000111111')	implies WORD, via 'ON'
(A,EQ,B,WORD)	specifies WORD
(A,EQ,B,DWORD)	specifies DWORD
((R3,6),NMIX,=X'000F')	implies WORD, via 'NMIX'
(FR0,GT,FR2,DFLOAT)	specifies double Floating (64 bits)
(R3,LGT,LABEL,ADDR)	specifies ADDRESS

Note. IF and DO macros permit logically-connected strings of statements. If a relational statement precedes a conditional statement, remember that the relational statement changes the LSR. If you want the conditional statement to test the LSR as you set it up in your preceding code, then arrange the statement string so that there are no intervening relational statements.

General Notes

- Structured programming macro generated code is reenterable.
- During execution, the generated code does NOT modify any floating-point registers, general purpose registers, or main storage, with the following exceptions:
  - The LSR indicators (negative, carry, overflow, and zero) are modified.
  - The CASEBLK's REG= register's value is doubled. (If END= and TREG= are both coded, then REG= register's value is quadrupled.) If coded, TREG=register's value is unpredictably modified.
  - During the DO-COUNT structure, a user-specified general-purpose register is decremented.
- Structured programming macros let the programmer use all hardware facilities (such as indicrect addressing) and data types (such as bit processing).
- Structured programming macros permit up to 20 levels of structure nesting.
- The macros print English-language assembly-time diagnostic messages. In many cases, these messages include the variables the programmer coded--this helps him find and correct his errors. (For more information reference the *Messages and Codes* manual, SC34-0126.

Examples

Examples of Compound Expressions

```
(A,EQ,B),AND,(C,GT,D)
EQ,OR,(E,LT,F)
((R1,10),MIX,=X'0807'),OR,OFF
(A,EQ,B),OR,(A,EQ,C),OR(A,EQ,D)
(A,LT,B),AND,EV       A less than B by an even difference
```

BLOCK Examples

```
ABC     BLOCK
        •
        BLOCK
          •
          LEAVE BLOCK=ABC
          •
        ENDBLK
DONE    ENDBLK   BLOCK=ABC
```

IF Examples

```
ABC       BLOCK
          •
          IF    (A,EQ,B),OR,((R2,4),IS,ON),END=FAR
            •
POSITIVE    IF    P,THEN
            •
            ELSE
            •
            ENDIF  IF=POSITIVE
            •
          ENDIF
          •
END       ENDBLK BLOCK=ABC
```

DO Example

```
START     BLOCK
          •
IFON      IF         Z,END=FAR
            •
LOOP1       DO       INF
              •
              DO      UNTIL,(A,EQ,B),OR,(A,EQ,C)
                LEAVE (R1,LE,R5),DO=LOOP1
                •
              ENDDO
ENDLOOP1    ENDDO  DO=LOOP1
              •
              •
          ELSE
            DO  WHILE,(A,NE,B),AND,(A,NE,C),OR,(R1,LE,R5)
              •
              •
              •
              •
              •
              •
              •
            ENDDO
          ENDIF  IF=IFON
          •
END       ENDBLK     BLOCK=START
```

Note. Because Boolean rules are *not* followed, a branch to the false code will occur even when the relation following the OR is *true* if the relation preceding the AND is false. See rules of expression evaluation under the IF macro section above.

CASE Examples

```
MAINPGM   BLOCK
          •
FAN       CASEBLK  REG=R1        REG1=0 THRU 5
            CASE 0,4
              •
            CASE 2,5
              DO   UNTIL,(R2,GT,XYZ)
                •
              ENDDO
            CASE 3               NULL CASE-NOTHING BETWEEN
                                 CASE & ENDCASE
          ENDCASE  CASEBLK=FAN
          •
MAINEND   ENDBLK   BLOCK=MAINPGM
```

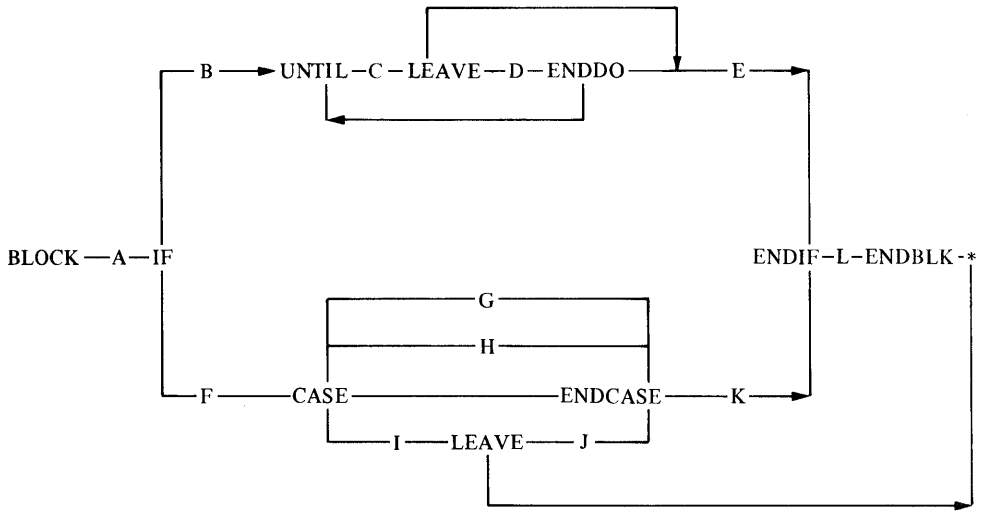
LEAVE Example

```
PGMA       BLOCK
           •
LOOP1      DO  WHILE,((R1,9),IS,ON,BIT)
             •
             IF   (R5,LT,XYZCT)
EXIT7F         LEAVE ((R4),EQ,=X'7F'),BLOCK=PGMA
             ELSE
               •
             ENDIF
ENDLOOP1   ENDDO   DO=LOOP1
           •
END        ENDBLK     BLOCK=PGMA
```

The following code represents an operator station interrupt service routine:

```
TTYINT     BLOCK
           code A
OK         IF  CC7,THEN
*INTERRUPT  NON-ERROR PROCESS
            code B
PACK        DO UNTIL,(R2,EQ,=X'7F'),OR,(A,GT,B*)
              code C
              LEAVE (CNT,GE,MAX),DO=PACK
              code D
            ENDDO DO=PACK
            code E
           ELSE
*INTERRUPT ERROR PROCESS
            code F
ERRFAN      CASEBLK REG=R1 CONDITION CODE IN REG 1
             CASE 0,3,4,END=FAR
               code G
             CASE 1,5
               code H
             CASE 2
               code I
             LEAVE (X,LLT,Y),BLOCK=TTYINT
               code J
            ENDCASE  CASEBLK=ERRFAN
            code K
           ENDIF   IF=OK
           code L
DONE       ENDBLK BLOCK=TTYINT
```

Schematic Example Of Structured Macro Use



Miscellaneous Notes on Internals

Labels

The programmer can code labels on structured macros. These labels are intended to be used as reference points during debugging and text-editing operations, and to maintain the structured concept, the programmer should not refer to them in assembler language code. Code generated by structured programming macros does not refer to a user's labels.

The following chart shows the assembler language labels that are generated by the structured programming macros:

```
MACRO     LABEL
BLOCK     --
ENDBLK    ##ENnnnn   target of LEAVE
IF        ##ITnnnn   target of true condition
ELSE      ##IFnnnn   target of false condition
ENDIF     ##IFnnnn   target of false condition
          ##ENnnnn   target of LEAVE
DO        ##DOnnnn   target of ENDDO
          ##DUnnnn   target of true condition
ENDDO     ##ENnnnn   target of LEAVE or false condition
CASEBLK   --
CASE      ##CSnnnn   target of CASEBLK
ENDCASE   ##CVnnnn   CASE vector table
          ##ENnnnn   target of LEAVE or CASE
```

Code Sizing

```
MACRO          WORDS
BLOCK     0
ENDBLK    0
IF        ((END+#WI)*#COMP)+#COND+2
ELSE      1+END
ENDIF     0
DO        ((END+#WI)*#COMP)+#COND+2
DO-INF    0
DO-COUNT  1
ENDDO     1+END
CASEBLK   3  (2 if TREG coded)
CASE      1+END-FIRST
ENDCASE   MAX+2  (MAX+4 if TREG coded on CASEBLK;
          2*MAX+4 if both TREG and END coded on CASEBLK)
LEAVE     1+END
THEN      0

Where:

END=0     if END=FAR not coded
   =1     if END=FAR coded
#WI=      number of words generated for immediate data elements
#COMP=    number of 'relationals' in compound expression
#COND=    number of 'conditionals' tested
MAX=      largest numeric coded on related CASE macros
FIRST=0   if first CASE
           =1  if not first CASE
```

Conditional-Setting Instructions

The following define which assembler instructions are generated by the structured programming macros for the purpose of evaluating a relational. The syntaxes permitted by the macros are directly governed by the assembler as explained in the "Relational Syntax" Section.

Note. For the following syntax, when you make a choice of operand format for one operand, you must choose the format in the same relative position in the other operand. For example, in CB, if you choose *addr5* for the first operand you must choose *addr4* for the other operand, *reg* is not allowed with *addr5*.

```
DATA      ASM
WIDTH     OPCODE    STRUCTURED MACRO RELATIONAL SYNTAX
BIT       TBT       ((reg,bitdisp),IS,{ON|NON|OFF|NOFF}[,BIT])
BYTE      CB        ({addr4|addr5},rel,{reg|addr4},BYTE)
          CBI       (reg,rel,immed8,BYTE)
WORD      CW        ({reg|addr4|addr5},rel,{reg|reg|addr4}
                    [,WORD])
          CWI       ({reg|addr4},rel,immed16[,WORD])
          TWI       {reg|addr4},{ON|NON },immed16[,WORD])
                       {OFF|NOFF }
                       {MIX|NMIX}
DWORD     CD        ({addr4|addr5},rel,{reg|addr4},DWORD)
FLOAT     FC        (fp-reg,rel,fp-reg,FLOAT)
DFLOAT    FCD       (fp-reg,rel,fp-reg,DFLOAT)
ADDR      CA        ({REG,ADDR4},rel,raddr,ADDR)
                    immed8:=C'.',=X'..',=B'........',
                    =expression,=S'.'
                    immed16:=C'..',=X'....',
                    =B................',
                    =expression,=S'..'
```

Reference Summary

```
[label]   BLOCK     [LEVELS={YES|NO}]
[label]   CASE      n[,n[,..]][,END=FAR]
[label]   CASEBLK   REG=reg[,TREG=reg][,END=FAR]
[label]   DO        INF[,END=FAR]
[label]   DO        CNTREG=reg
[label]   DO        UNTIL,{statement}[,{AND|OR},{statement}
          [,..]][,END=FAR]
[label]   DO        WHILE,{statement}[,{AND|OR},{statement}
          [,..]][,END=FAR]
[label]   ELSE      [END=FAR]
[label]   ENDBLK    [BLOCK=label]
[label]   ENDCASE   [CASEBLK=label]
[label]   ENDDO     [DO=label]
[label]   ENDIF     [IF=label]
[label]   IF        {statement}[,{AND|OR},{statement}[,..]]
          [,END=FAR][,THEN]
[label]   LEAVE     [{statement},]{BLOCK|IF|DO|CASEBLK}
          =label[,END=FAR]
          THEN
```

CONDITIONALS		RELATIONAL	CONDITIONS		WIDTHS
EQ	NE	LOGICAL:	EQ	NE	BIT
P	NP		LGT	LLT	BYTE
N	NN		LGE	LLE	WORD
EV	NEV	ARITHMETIC:	GT	LT	DWORD
Z	NZ		GE	LE	FLOAT
CY	NCY	BIT MASK:	ON	NON	DFLOAT
OV	NOV		OFF	NOFF	ADDR
ON	NON				
OFF	NOFF				
MIX	NMIX				
LGT	LLE				
LGE	LLT				
GT	LE				
GE	LT				
CCn	NCCn				
ER	NER				

```
(p1,rel,p2[,width])
(p1,IS,{ON|NON|OFF|NOFF}[,BIT])
```

Structured Macro Generated Message Format

Note. For more information reference the *Messages and Codes* manual, SC34-0126.

ERRORS/WARNINGS: Messages are parameterized with operand values that are in error.

```
MNOTE     m,'CCCnn***(ERROR WARNING):----------'
MNOTE     m,'CCCnn***ACTION:--------------------'
```

Examples:

```
MNOTE     4,'CPB01***ERROR:  "&OP1" BEYOND MAX "&CNTS"'
MNOTE     *,'CPB02***ACTION: DEFAULT "CNT=128"'
```

DEFAULTS of optional operands:

```
MNOTE     *,'CCCnn***DEFAULT:--------------------'
```

Example:

```
MNOTE     *,'CPB04***DEFAULT"  BLKSIZE=128 BYTES '
MNOTE m   CCCnn
          m=*   INFORMATION, COMMENT IS PRINTED
          m=4   MINOR ERRORS DETECTED; SUCCESSFUL PROGRAM
                EXECUTION IS PROBABLE.
          m=8   ERRORS DETECTED, UNSUCCESSFUL PROGRAM
                EXECUTION IS POSSIBLE.
          m=12  SERIOUS ERRORS DETECTED; UNSUCCESSFUL
                EXECUTION IS PROBABLE.
          m=16  CRITICAL ERRORS DETECTED: NORMAL EXECUTION
                IS IMPOSSIBLE
          CCC   COMPONENT CODE
          nn    DECIMAL NUMBER ID OF MESSAGE
```

Appendix B. Decimal/Binary/Hexadecimal Conversions

DECIMAL TO BINARY CONVERSION

Several methods exist for converting binary numbers to decimal numbers. The method used here is repetitive division. To find the binary equivalent of a decimal number:

1. Divide the decimal number by 2.
2. Save the remainder (0 or 1).
3. Divide the quotient by 2.
4. Repeat steps 2 and 3 until the quotient can no longer be divided. The last quotient is the last remainder.

Example: Convert decimal 236 to binary.

$2\overline{)236}$ R = 0

$2\overline{)118}$ R = 0

$2\overline{)59}$ R = 1

$2\overline{)29}$ R = 1

$2\overline{)14}$ R = 0

$2\overline{)7}$ R = 1

$2\overline{)3}$ R = 1

R = 1 Last quotient

To represent this binary number in a byte: first, assign the units position of the binary number the value 1, the tens position the value 2, and the hundreds position the value 4. Doubling the value each time, assign values to all bit positions through the high-order bit as shown in the following example:

Bit position	0	1	2	3	4	5	6	7
Decimal assigned values	128	64	32	16	8	4	2	1

Second, take the remainders you obtained in the division process and place them in each bit position. Place the first remainder in bit position 7. Fill all unused high-order bits with 0.

Bit position	0	1	2	3	4	5	6	7
Decimal assigned value	128	64	32	16	8	4	2	1
Binary value of 236	1	1	1	0	1	1	0	0

BINARY TO DECIMAL CONVERSION

To convert from binary to decimal: add the decimal equivalent of each bit position that contains 1. In this example, binary 1110 1100 = 128 + 64 + 32 + 8 + 4 = 236 decimal.

BINARY TO HEXADECIMAL CONVERSION

Hexadecimal numbering is similar to decimal numbering. However, since the hexadecimal base is 16, numbers greater than 9 are assigned alphabetic equivalents, as follows:

Decimal	*Hexadecimal*
0	0
1	1
2	2
3	3
4	4
5	5
6	6
7	7
8	8
9	9
10	A
11	B
12	C
13	D
14	E
15	F

Four binary bits represent a hexadecimal number.

Binary	*Hexadecimal*
0000	0
0001	1
0010	2
0011	3
0100	4
0101	5
0110	6
0111	7
1000	8
1001	9
1010	A
1011	B
1100	C
1101	D
1110	E
1111	F

Two hexadecimal numbers are represented in a byte. Bit positions 0–3 represent the first hexadecimal character, and bit positions 4–7 represent the second hexadecimal character. The following examples show the hexadecimal equivalents of one-byte binary numbers.

	One-byte binary number								Hexadecimal equivalent	
									First character	Second character
Bit position	0	1	2	3	4	5	6	7		
Examples	0	0	0	1	1	0	0	0	1	8
	0	0	1	0	0	0	0	0	2	0
	0	0	1	1	0	0	1	0	3	2
	0	1	0	0	1	1	0	0	4	C
	1	1	1	1	1	1	1	0	F	E

Figure B-1 gives the binary, decimal, and hexadecimal equivalents from 0 to 255. Where applicable, equivalent printer graphics are also shown.

Binary				
First half byte 128 64 32 16 8421	Second half byte 8421	Printer graphic	Decimal	Hexa-decimal
0000	0000		0	00
0000	0001		1	01
0000	0010		2	02
0000	0011		3	03
0000	0100		4	04
0000	0101		5	05
0000	0110		6	06
0000	0111		7	07
0000	1000		8	08
0000	1001		9	09
0000	1010		10	0A
0000	1011		11	0B
0000	1100		12	0C
0000	1101		13	0D
0000	1110		14	0E
0000	1111		15	0F
0001	0000		16	10
0001	0001		17	11
0001	0010		18	12
0001	0011		19	13
0001	0100		20	14
0001	0101		21	15
0001	0110		22	16
0001	0111		23	17
0001	1000		24	18
0001	1001		25	19
0001	1010		26	1A
0001	1011		27	1B
0001	1100		28	1C
0001	1101		29	1D
0001	1110		30	1E
0001	1111		31	1F
0010	0000		32	20
0010	0001		33	21
0010	0010		34	22
0010	0011		35	23
0010	0100		36	24
0010	0101		37	25
0010	0110		38	26
0010	0111		39	27
0010	1000		40	28
0010	1001		41	29
0010	1010		42	2A
0010	1011		43	2B
0010	1100		44	2C
0010	1101		45	2D
0010	1110		46	2E
0010	1111		47	2F
0011	0000		48	30
0011	0001		49	31
0011	0010		50	32
0011	0011		51	33
0011	0100		52	34
0011	0101		53	35
0011	0110		54	36
0011	0111		55	37
0011	1000		56	38
0011	1001		57	39
0011	1010		58	3A
0011	1011		59	3B
0011	1100		60	3C
0011	1101		61	3D
0011	1110		62	3E
0011	1111		63	3F
0100	0000	blank	64	40

Binary				
First half byte 128 64 32 16 8421	Second half byte 8421	Printer graphic	Decimal	Hexa-decimal
0100	0001		65	41
0100	0010		66	42
0100	0011		67	43
0100	0100		68	44
0100	0101		69	45
0100	0110		70	46
0100	0111		71	47
0100	1000		72	48
0100	1001		73	49
0100	1010		74	4A
0100	1011	. (period)	75	4B
0100	1100	<	76	4C
0100	1101	(	77	4D
0100	1110	+	78	4E
0100	1111		79	4F
0101	0000	&	80	50
0101	0001		81	51
0101	0010		82	52
0101	0011		83	53
0101	0100		84	54
0101	0101		85	55
0101	0110		86	56
0101	0111		87	57
0101	1000		88	58
0101	1001		89	59
0101	1010		90	5A
0101	1011	$	91	5B
0101	1100	*	92	5C
0101	1101	)	93	5D
0101	1110		94	5E
0101	1111		95	5F
0110	0000	–	96	60
0110	0001	/	97	61
0110	0010		98	62
0110	0011		99	63
0110	0100		100	64
0110	0101		101	65
0110	0110		102	66
0110	0111		103	67
0110	1000		104	68
0110	1001		105	69
0110	1010		106	6A
0110	1011	,	107	6B
0110	1100	%	108	6C
0110	1101		109	6D
0110	1110		110	6E
0110	1111		111	6F
0111	0000		112	70
0111	0001		113	71
0111	0010		114	72
0111	0011		115	73
0111	0100		116	74
0111	0101		117	75
0111	0110		118	76
0111	0111		119	77
0111	1000		120	78
0111	1001		121	79
0111	1010	:	122	7A
0111	1011	#	123	7B
0111	1100	@	124	7C
0111	1101	'	125	7D
0111	1110	=	126	7E
0111	1111		127	7F
1000	0000		128	80
1000	0001	a	129	81

Figure B-1 (Part 1 of 2). EBCDIC, hexadecimal, decimal table

Binary		Printer graphic	Decimal	Hexa-decimal
First half byte 128 64 32 16 8421	Second half byte 8421			
1000	0010	b	130	82
1000	0011	c	131	83
1000	0100	d	132	84
1000	0101	e	133	85
1000	0110	f	134	86
1000	0111	g	135	87
1000	1000	h	136	88
1000	1001	i	137	89
1000	1010		138	8A
1000	1011		139	8B
1000	1100		140	8C
1000	1101		141	8D
1000	1110		142	8E
1000	1111		143	8F
1001	0000		144	90
1001	0001	j	145	91
1001	0010	k	146	92
1001	0011	l	147	93
1001	0100	m	148	94
1001	0101	n	149	95
1001	0110	o	150	96
1001	0111	p	151	97
1001	1000	q	152	98
1001	1001	r	153	99
1001	1010		154	9A
1001	1011		155	9B
1001	1100		156	9C
1001	1101		157	9D
1001	1110		158	9E
1001	1111		159	9F
1010	0000		160	A0
1010	0001		161	A1
1010	0010	s	162	A2
1010	0011	t	163	A3
1010	0100	u	164	A4
1010	0101	v	165	A5
1010	0110	w	166	A6
1010	0111	x	167	A7
1010	1000	y	168	A8
1010	1001	z	169	A9
1010	1010		170	AA
1010	1011		171	AB
1010	1100		172	AC
1010	1101		173	AD
1010	1110		174	AE
1010	1111		175	AF
1011	0000		176	B0
1011	0001		177	B1
1011	0010		178	B2
1011	0011		179	B3
1011	0100		180	B4
1011	0101		181	B5
1011	0110		182	B6
1011	0111		183	B7
1011	1000		184	B8
1011	1001		185	B9
1011	1010		186	BA
1011	1011		187	BB
1011	1100		188	BC
1011	1101		189	BD
1011	1110		190	BE
1011	1111		191	BF
1100	0000		192	C0

Binary		Printer graphic	Decimal	Hexa-decimal
First half byte 128 64 32 16 8421	Second half byte 8421			
1100	0001	A	193	C1
1100	0010	B	194	C2
1100	0011	C	195	C3
1100	0100	D	196	C4
1100	0101	E	197	C5
1100	0110	F	198	C6
1100	0111	G	199	C7
1100	1000	H	200	C8
1100	1001	I	201	C9
1100	1010		202	CA
1100	1011		203	CB
1100	1100		204	CC
1100	1101		205	CD
1100	1110		206	CE
1100	1111		207	CF
1101	0000		208	D0
1101	0001	J	209	D1
1101	0010	K	210	D2
1101	0011	L	211	D3
1101	0100	M	212	D4
1101	0101	N	213	D5
1101	0110	O	214	D6
1101	0111	P	215	D7
1101	1000	Q	216	D8
1101	1001	R	217	D9
1101	1010		218	DA
1101	1011		219	DB
1101	1100		220	DC
1101	1101		221	DD
1101	1110		222	DE
1101	1111		223	DF
1110	0000		224	E0
1110	0001		225	E1
1110	0010	S	226	E2
1110	0011	T	227	E3
1110	0100	U	228	E4
1110	0101	V	229	E5
1110	0110	W	230	E6
1110	0111	X	231	E7
1110	1000	Y	232	E8
1110	1001	Z	233	E9
1110	1010		234	EA
1110	1011		235	EB
1110	1100		236	EC
1110	1101		237	ED
1110	1110		238	EE
1110	1111		239	EF
1111	0000	0	240	F0
1111	0001	1	241	F1
1111	0010	2	242	F2
1111	0011	3	243	F3
1111	0100	4	244	F4
1111	0101	5	245	F5
1111	0110	6	246	F6
1111	0111	7	247	F7
1111	1000	8	248	F8
1111	1001	9	249	F9
1111	1010		250	FA
1111	1011		251	FB
1111	1100		252	FC
1111	1101		253	FD
1111	1110		254	FE
1111	1111		255	FF

Figure B-1 (Part 2 of 2). EBCDIC, hexadecimal, decimal table

Hexadecimal/Decimal Conversion

Figure B-2 provides direct conversion of decimal and hexadecimal numbers in these ranges:

Hexadecimal	*Decimal*
000 to FFF	0000 to 4095

Decimal numbers are within the table. The first two *hexadecimal* characters are in the left column of the table; the third hexadecimal character (x) is arranged across the top of the table.

For numbers outside the range of the table, add the following values to the table figures.

Hexadecimal	*Decimal*
1000	4096
2000	8192
3000	12288
4000	16384
5000	20480
6000	24576
7000	28672
8000	32768
9000	36864
A000	40960
B000	45056
C000	49152
D000	53248
E000	57344
F000	61440

DECIMAL TO HEXADECIMAL

To convert from decimal to hexadecimal using Figure B-2:

1. Find the decimal number in the table.
2. Determine the hexadecimal number.
   a. Locate the hexadecimal characters in the left column.
   b. Substitute the value for x (the character in the top column above the decimal number).

Example: Decimal 123 is equivalent to hexadecimal 07B; decimal 1478 is equivalent to hexadecimal 5C6.

HEXADECIMAL TO DECIMAL

To find the decimal equivalent of a hexadecimal number using Figure B-2:

1. Find the first two hexadecimal characters in the left column.
2. Scan across this row until you find the column containing the last hexadecimal character. Here is the decimal number.

Example: Find decimal equivalent of hexadecimal 0C9. Find 0C in the left column. Look under column 9. The decimal number is 0201.

x = 0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
00x	0000	0001	0002	0003	0004	0005	0006	0007	0008	0009	0010	0011	0012	0013	0014	0015
01x	0016	0017	0018	0019	0020	0021	0022	0023	0024	0025	0026	0027	0028	0029	0030	0031
02x	0032	0033	0034	0035	0036	0037	0038	0039	0040	0041	0042	0043	0044	0045	0046	0047
03x	0048	0049	0050	0051	0052	0053	0054	0055	0056	0057	0058	0059	0060	0061	0062	0063
04x	0064	0065	0066	0067	0068	0069	0070	0071	0072	0073	0074	0075	0076	0077	0078	0079
05x	0080	0081	0082	0083	0084	0085	0086	0087	0088	0089	0090	0091	0092	0093	0094	0095
06x	0096	0097	0098	0099	0100	0101	0102	0103	0104	0105	0106	0107	0108	0109	0110	0111
07x	0112	0113	0114	0115	0116	0117	0118	0119	0120	0121	0122	0123	0124	0125	0126	0127
08x	0128	0129	0130	0131	0132	0133	0134	0135	0136	0137	0138	0139	0140	0141	0142	0143
09x	0144	0145	0146	0147	0148	0149	0150	0151	0152	0153	0154	0155	0156	0157	0158	0159
0Ax	0160	0161	0162	0163	0164	0165	0166	0167	0168	0169	0170	0171	0172	0173	0174	0175
0Bx	0176	0177	0178	0179	0180	0181	0182	0183	0184	0185	0186	0187	0188	0189	0190	0191
0Cx	0192	0193	0194	0195	0196	0197	0198	0199	0200	0201	0202	0203	0204	0205	0206	0207
0Dx	0208	0209	0210	0211	0212	0213	0214	0215	0216	0217	0218	0219	0220	0221	0222	0223
0Ex	0224	0225	0226	0227	0228	0229	0230	0231	0232	0233	0234	0235	0236	0237	0238	0239
0Fx	0240	0241	0242	0243	0244	0245	0246	0247	0248	0249	0250	0251	0252	0253	0254	0255
10x	0256	0257	0258	0259	0260	0261	0262	0263	0264	0265	0266	0267	0268	0269	0270	0271
11x	0272	0273	0274	0275	0276	0277	0278	0279	0280	0281	0282	0283	0284	0285	0286	0287
12x	0288	0289	0290	0291	0292	0293	0294	0295	0296	0297	0298	0299	0300	0301	0302	0303
13x	0304	0305	0306	0307	0308	0309	0310	0311	0312	0313	0314	0315	0316	0317	0318	0319
14x	0320	0321	0322	0323	0324	0325	0326	0327	0328	0329	0330	0331	0332	0333	0334	0335
15x	0336	0337	0338	0339	0340	0341	0342	0343	0344	0345	0346	0347	0348	0349	0350	0351
16x	0352	0353	0354	0355	0356	0357	0358	0359	0360	0361	0362	0363	0364	0365	0366	0367
17x	0368	0369	0370	0371	0372	0373	0374	0375	0376	0377	0378	0379	0380	0381	0382	0383
18x	0384	0385	0386	0387	0388	0389	0390	0391	0392	0393	0394	0395	0396	0397	0398	0399
19x	0400	0401	0402	0403	0404	0405	0406	0407	0408	0409	0410	0411	0412	0413	0414	0415
1Ax	0416	0417	0418	0419	0420	0421	0422	0423	0424	0425	0426	0427	0428	0429	0430	0431
1Bx	0432	0433	0434	0435	0436	0437	0438	0439	0440	0441	0442	0443	0444	0445	0446	0447
1Cx	0448	0449	0450	0451	0452	0453	0454	0455	0456	0457	0458	0459	0460	0461	0462	0463
1Dx	0464	0465	0466	0467	0468	0469	0470	0471	0472	0473	0474	0475	0476	0477	0478	0479
1Ex	0480	0481	0482	0483	0484	0485	0486	0487	0488	0489	0490	0491	0492	0493	0494	0495
1Fx	0496	0497	0498	0499	0500	0501	0502	0503	0504	0505	0506	0507	0508	0509	0510	0511
20x	0512	0513	0514	0515	0516	0517	0518	0519	0520	0521	0522	0523	0524	0525	0526	0527
21x	0528	0529	0530	0531	0532	0533	0534	0535	0536	0537	0538	0539	0540	0541	0542	0543
22x	0544	0545	0546	0547	0548	0549	0550	0551	0552	0553	0554	0555	0556	0557	0558	0559
23x	0560	0561	0562	0563	0564	0565	0566	0567	0568	0569	0570	0571	0572	0573	0574	0575
24x	0576	0577	0578	0579	0580	0581	0582	0583	0584	0585	0586	0587	0588	0589	0590	0591
25x	0592	0593	0594	0595	0596	0597	0598	0599	0600	0601	0602	0603	0604	0605	0606	0607
26x	0608	0609	0610	0611	0612	0613	0614	0615	0616	0617	0618	0169	0620	0621	0622	0623
27x	0624	0625	0626	0627	0628	0269	0630	0631	0632	0633	0634	0635	0636	0637	0638	0639
28x	0640	0641	0642	0643	0644	0645	0646	0647	0648	0649	0650	0651	0652	0653	0654	0655
29x	0656	0657	0658	0659	0660	0661	0662	0663	0664	0665	0666	0667	0668	0669	0670	0671
2Ax	0672	0673	0674	0675	0676	0677	0678	0679	0680	0681	0682	0683	0684	0685	0686	0687
2Bx	0688	0689	0690	0691	0692	0693	0694	0695	0696	0697	0698	0699	0700	0701	0702	0703
2Cx	0704	0705	0706	0707	0708	0709	0710	0711	0712	0713	0714	0715	0716	0717	0718	0719
2Dx	0720	0721	0722	0723	0724	0725	0726	0727	0728	0729	0730	0731	0732	0733	0734	0735
2Ex	0736	0737	0738	0739	0740	0741	0742	0743	0744	0745	0746	0747	0748	0749	0750	0751
2Fx	0752	0753	0754	0755	0756	0757	0758	0759	0760	0761	0762	0763	0764	0765	0766	0767
30x	0768	0769	0770	0771	0772	0773	0774	0775	0776	0777	0778	0779	0780	0781	0782	0783
31x	0784	0785	0786	0787	0788	0789	0790	0791	0792	0793	0794	0795	0796	0797	0798	0799
32x	0800	0801	0802	0803	0804	0805	0806	0807	0808	0809	0810	0811	0812	0813	0814	0815
33x	0816	0817	0818	0819	0820	0821	0822	0823	0824	0825	0826	0827	0828	0829	0830	0831
34x	0832	0833	0834	0835	0836	0837	0838	0839	0840	0841	0842	0843	0844	0845	0846	0847
35x	0848	0849	0850	0851	0852	0853	0854	0855	0856	0857	0858	0859	0860	0861	0862	0863
36x	0864	0865	0866	0867	0868	0869	0870	0871	0872	0873	0874	0875	0876	0877	0878	0879
37x	0880	0881	0882	0883	0884	0885	0886	0887	0888	0889	0890	0891	0892	0893	0894	0895
38x	0896	0897	0898	0899	0900	0901	0902	0903	0904	0905	0906	0907	0908	0909	0910	0911
39x	0912	0913	0914	0915	0916	0917	0918	0919	0920	0921	0922	0923	0924	0925	0926	0927
3Ax	0928	0929	0930	0931	0932	0933	0934	0935	0936	0937	0938	0939	0940	0941	0942	0943
3Bx	0944	0945	0946	0947	0948	0949	0950	0951	0952	0953	0954	0955	0956	0957	0958	0959
3Cx	0960	0961	0962	0963	0964	0965	0966	0967	0968	0969	0970	0971	0972	0973	0974	0975
3Dx	0976	0977	0978	0979	0980	0981	0982	0983	0984	0985	0986	0987	0988	0989	0990	0991
3Ex	0992	0993	0994	0995	0996	0997	0998	0999	1000	1001	1002	1003	1004	1005	1006	1007
3Fx	1008	1009	1010	1011	1012	1013	1014	1015	1016	1017	1018	1019	1020	1021	1022	1023

Figure B-2 (Part 1 of 4). Hexadecimal/decimal conversion table

x = 0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
40x	1024	1025	1026	1027	1028	1029	1030	1031	1032	1033	1034	1035	1036	1037	1038	1039
41x	1040	1041	1042	1043	1044	1045	1046	1047	1048	1049	1050	1051	1052	1053	1054	1055
42x	1056	1057	1058	1059	1060	1061	1062	1063	1064	1065	1066	1067	1068	1069	1070	1071
43x	1072	1073	1074	1075	1076	1077	1078	1079	1080	1081	1082	1083	1084	1085	1086	1087
44x	1088	1089	1090	1091	1092	1093	1094	1095	1096	1097	1098	1099	1100	1101	1102	1103
45x	1104	1105	1106	1107	1108	1109	1110	1111	1112	1113	1114	1115	1116	1117	1118	1119
46x	1120	1121	1122	1123	1124	1125	1126	1127	1128	1129	1130	1131	1132	1133	1134	1135
47x	1136	1137	1138	1139	1140	1141	1142	1143	1144	1145	1146	1147	1148	1149	1150	1151
48x	1152	1153	1154	1155	1156	1157	1158	1159	1160	1161	1162	1163	1164	1165	1166	1167
49x	1168	1169	1170	1171	1172	1173	1174	1175	1176	1177	1178	1179	1180	1181	1182	1183
4Ax	1184	1185	1186	1187	1188	1189	1190	1191	1192	1193	1194	1195	1196	1197	1198	1199
4Bx	1200	1201	1202	1203	1204	1205	1206	1207	1208	1209	1210	1211	1212	1213	1214	1215
4Cx	1216	1217	1218	1219	1220	1221	1222	1223	1224	1225	1226	1227	1228	1229	1230	1231
4Dx	1232	1233	1234	1235	1236	1237	1238	1239	1240	1241	1242	1243	1244	1245	1246	1247
4Ex	1248	1249	1250	1251	1252	1253	1254	1255	1256	1257	1258	1259	1260	1261	1262	1263
4Fx	1264	1265	1266	1267	1268	1269	1270	1271	1272	1273	1274	1275	1276	1277	1278	1279
50x	1280	1281	1282	1283	1284	1285	1286	1287	1288	1289	1290	1291	1292	1293	1294	1295
51x	1296	1297	1298	1299	1300	1301	1302	1303	1304	1305	1306	1307	1308	1309	1310	1311
52x	1312	1313	1314	1315	1316	1317	1318	1319	1320	1321	1322	1323	1324	1325	1326	1327
53x	1328	1329	1330	1331	1332	1333	1334	1335	1336	1337	1338	1339	1340	1341	1342	1343
54x	1344	1345	1346	1347	1348	1349	1350	1351	1352	1353	1354	1355	1356	1357	1358	1359
55x	1360	1361	1362	1363	1364	1365	1366	1367	1368	1369	1370	1371	1372	1373	1374	1375
56x	1376	1377	1378	1379	1380	1381	1382	1383	1384	1385	1386	1387	1388	1389	1390	1391
57x	1392	1393	1394	1395	1396	1397	1398	1399	1400	1401	1402	1403	1404	1405	1406	1407
58x	1408	1409	1410	1411	1412	1413	1414	1415	1416	1417	1418	1419	1420	1421	1422	1423
59x	1424	1425	1426	1427	1428	1429	1430	1431	1432	1433	1434	1435	1436	1437	1438	1439
5Ax	1440	1441	1442	1443	1444	1445	1446	1447	1448	1449	1450	1451	1452	1453	1454	1455
5Bx	1456	1457	1458	1459	1460	1461	1462	1463	1464	1465	1466	1467	1468	1469	1470	1471
5Cx	1472	1473	1474	1475	1476	1477	1478	1479	1480	1481	1482	1483	1484	1485	1486	1487
5Dx	1488	1489	1490	1491	1492	1493	1494	1495	1496	1497	1498	1499	1500	1501	1502	1503
5Ex	1504	1505	1506	1507	1508	1509	1510	1511	1512	1513	1514	1515	1516	1517	1518	1519
5Fx	1520	1521	1522	1523	1524	1525	1526	1527	1528	1529	1530	1531	1532	1533	1534	1535
60x	1536	1537	1538	1539	1540	1541	1542	1543	1544	1545	1546	1547	1548	1549	1550	1551
61x	1552	1553	1554	1555	1556	1557	1558	1559	1560	1561	1562	1563	1564	1565	1566	1567
62x	1568	1569	1570	1571	1572	1573	1574	1575	1576	1577	1578	1579	1580	1581	1582	1583
63x	1584	1585	1586	1587	1588	1589	1590	1591	1592	1593	1594	1595	1596	1597	1598	1599
64x	1600	1601	1602	1603	1604	1605	1606	1607	1608	1609	1610	1611	1612	1613	1614	1615
65x	1616	1617	1618	1619	1620	1621	1622	1623	1624	1625	1626	1627	1628	1629	1630	1631
66x	1632	1633	1634	1635	1636	1637	1638	1639	1640	1641	1642	1643	1644	1645	1646	1647
67x	1648	1649	1650	1651	1652	1653	1654	1655	1656	1657	1658	1659	1660	1661	1662	1663
68x	1664	1665	1666	1667	1668	1699	1670	1671	1672	1673	1674	1675	1676	1677	1678	1679
69x	1680	1681	1682	1683	1684	1685	1686	1687	1688	1689	1690	1691	1692	1693	1694	1695
6Ax	1696	1697	1698	1699	1700	1701	1702	1703	1704	1705	1706	1707	1708	1709	1710	1711
6Bx	1712	1713	1714	1715	1716	1717	1718	1719	1720	1721	1722	1723	1724	1725	1726	1727
6Cx	1728	1729	1730	1731	1732	1733	1734	1735	1736	1737	1738	1739	1740	1741	1742	1743
6Dx	1744	1745	1746	1747	1748	1749	1750	1751	1752	1753	1754	1755	1756	1757	1758	1759
6Ex	1760	1761	1762	1763	1764	1765	1766	1767	1768	1769	1770	1771	1772	1773	1774	1775
6Fx	1776	1777	1778	1779	1780	1781	1782	1783	1784	1785	1786	1787	1788	1789	1790	1791
70x	1792	1793	1794	1795	1796	1797	1798	1799	1800	1801	1802	1803	1804	1805	1806	1807
71x	1808	1809	1810	1811	1812	1813	1814	1815	1816	1817	1818	1819	1820	1821	1822	1823
72x	1824	1825	1826	1827	1828	1829	1830	1831	1832	1833	1834	1835	1836	1837	1838	1839
73x	1840	1841	1842	1843	1844	1845	1846	1847	1848	1849	1850	1851	1852	1853	1854	1855
74x	1856	1857	1858	1859	1860	1861	1862	1863	1864	1865	1866	1867	1868	1869	1870	1871
75x	1872	1873	1874	1875	1876	1877	1878	1879	1880	1881	1882	1883	1884	1885	1886	1887
76x	1888	1889	1890	1891	1892	1893	1894	1895	1896	1897	1898	1899	1900	1901	1902	1903
77x	1904	1905	1906	1907	1908	1909	1910	1911	1912	1913	1914	1915	1916	1917	1918	1919
78x	1920	1921	1922	1923	1924	1925	1926	1927	1928	1929	1930	1931	1932	1933	1934	1935
79x	1936	1937	1938	1939	1940	1941	1942	1943	1944	1945	1946	1947	1948	1949	1950	1951
7Ax	1952	1953	1954	1955	1956	1957	1958	1959	1960	1961	1962	1963	1964	1965	1966	1976
7Bx	1968	1969	1970	1971	1972	1973	1974	1975	1976	1977	1978	1979	1980	1981	1982	1983
7Cx	1984	1985	1986	1987	1988	1989	1990	1991	1992	1993	1994	1995	1996	1997	1998	1999
7Dx	2000	2001	2002	2003	2004	2005	2006	2007	2008	2009	2010	2011	2012	2013	2014	2015
7Ex	2016	2017	2018	2019	2020	2021	2022	2023	2024	2025	2026	2027	2028	2029	2030	2031
7Fx	2032	2033	2034	2035	2036	2037	2038	2039	2040	2041	2042	2043	2044	2045	2046	2047

Figure B-2 (Part 2 of 4). Hexadecimal/decimal conversion table

x = 0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
80x	2048	2049	2050	2051	2052	2053	2054	2055	2056	2057	2058	2059	2060	2061	2062	2063
81x	2064	2065	2066	2067	2068	2069	2070	2071	2072	2073	2074	2075	2076	2077	2078	2079
82x	2080	2081	2082	2083	2084	2085	2086	2087	2088	2089	2090	2091	2092	2093	2094	2095
83x	2096	2097	2098	2099	2100	2101	2102	2103	2104	2105	2106	2107	2108	2109	2110	2111
84x	2112	2113	2114	2115	2116	2117	2118	2119	2120	2121	2122	2123	2124	2125	2126	2127
85x	2128	2129	2130	2131	2132	2133	2134	2135	2136	2137	2138	2139	2140	2141	2142	2143
86x	2144	2145	2146	2147	2148	2149	2150	2151	2152	2153	2154	2155	2156	2157	2158	2159
87x	2160	2161	2162	2163	2164	2165	2166	2167	2168	2169	2170	2171	2172	2173	2174	2175
88x	2176	2177	2178	2179	2180	2181	2182	2183	2184	2185	2186	2187	2188	2189	2190	2191
89x	2192	2193	2194	2195	2196	2197	2198	2199	2200	2201	2202	2203	2204	2205	2206	2207
8Ax	2208	2209	2210	2211	2212	2213	2214	2215	2216	2217	2218	2219	2220	2221	2222	2223
8Bx	2224	2225	2226	2227	2228	2229	2230	2231	2232	2233	2234	2235	2236	2237	2238	2239
8Cx	2240	2241	2242	2243	2244	2245	2246	2247	2248	2249	2250	2251	2252	2253	2254	2255
8Dx	2256	2257	2258	2259	2260	2261	2262	2263	2264	2265	2266	2267	2268	2269	2270	2271
8Ex	2272	2273	2274	2275	2276	2277	2278	2279	2280	2281	2282	2283	2284	2285	2286	2287
8Fx	2288	2289	2290	2291	2292	2293	2294	2295	2296	2297	2298	2299	2300	2301	2302	2303
90x	2304	2305	2306	2307	2308	2309	2310	2311	2312	2313	2314	2315	2316	2317	2318	2319
91x	2320	2321	2322	2323	2324	2325	2326	2327	2328	2329	2330	2331	2332	2333	2334	2335
92x	2336	2337	2338	2339	2340	2341	2342	2343	2344	2345	2346	2347	2348	2349	2350	2351
93x	2352	2353	2354	2355	2356	2357	2358	2359	2360	2361	2362	2363	2364	2365	2366	2367
94x	2368	2369	2370	2371	2372	2373	2374	2375	2376	2377	2378	2379	2380	2381	2382	2383
95x	2384	2385	2386	2387	2388	2389	2390	2391	2392	2393	2394	2395	2396	2397	2398	2399
96x	2400	2401	2402	2403	2404	2405	2406	2407	2408	2409	2410	2411	2412	2413	2414	2415
97x	2416	2417	2418	2419	2420	2421	2422	2423	2424	2425	2426	2427	2428	2429	2430	2431
98x	2432	2433	2434	2435	2436	2437	2438	2439	2440	2441	2442	2443	2444	2445	2446	2447
99x	2448	2449	2450	2451	2452	2453	2454	2455	2456	2457	2458	2459	2460	2471	2462	2463
9Ax	2464	2465	2466	2467	2468	2469	2470	2471	2472	2473	2474	2475	2476	2477	2478	2479
9Bx	2480	2481	2482	2483	2484	2485	2486	2487	2488	2489	2490	2491	2492	2493	2494	2495
9Cx	2496	2497	2498	2499	2500	2501	2502	2503	2504	2505	2506	2507	2508	2509	2510	2511
9Dx	2512	2513	2514	2515	2516	2517	2518	2519	2520	2521	2522	2523	2524	2525	2526	2527
9Ex	2528	2529	2530	2531	2532	2533	2534	2535	2536	2537	2538	2539	2540	2541	2542	2543
9Fx	2544	2545	2546	2547	2548	2549	2550	2551	2552	2553	2554	2555	2556	2557	2558	2559
A0x	2560	2561	2562	2563	2564	2565	2566	2567	2568	2569	2570	2571	2572	2573	2574	2575
A1x	2576	2577	2578	2579	2580	2581	2582	2583	2584	2585	2586	2587	2588	2589	2590	2591
A2x	2592	2593	2594	2595	2596	2597	2598	2599	2600	2601	2602	2603	2604	2605	2606	2607
A3x	2608	2609	2610	2611	2612	2613	2614	2615	2616	2617	2618	2619	2620	2621	2622	2623
A4x	2624	2625	2626	2627	2628	2629	2630	2631	2632	2633	2634	2635	2636	2637	2638	2639
A5x	2640	2641	2642	2643	2644	2645	2646	2647	2648	2649	2650	2651	2652	2653	2654	2655
A6x	2656	2657	2658	2659	2660	2661	2662	2663	2664	2665	2666	2667	2668	2699	2670	2671
A7x	2672	2673	2674	2675	2676	2677	2678	2679	2680	2681	2682	2683	2684	2685	2686	2687
A8x	2688	2689	2690	2691	2692	2693	2694	2695	2696	2697	2698	2699	2700	2701	2702	2703
A9x	2704	2705	2706	2707	2708	2709	2710	2711	2712	2713	2714	2715	2716	2717	2718	2719
AAx	2720	2721	2722	2723	2724	2725	2726	2727	2728	2729	2730	2731	2732	2733	2734	2735
ABx	2736	2737	2738	2739	2740	2741	2742	2743	2744	2745	2746	2747	2748	2749	2750	2751
ACx	2752	2753	2754	2755	2756	2757	2758	2759	2760	2761	2762	2763	2764	2765	2766	2767
ADx	2768	2769	2770	2771	2772	2773	2774	2775	2776	2777	2778	2779	2780	2781	2782	2783
AEx	2784	2785	2786	2787	2788	2789	2790	2791	2792	2793	2794	2795	2796	2797	2798	2799
AFx	2800	2801	2802	2803	2804	2805	2806	2807	2808	2809	2810	2811	2812	2813	2814	2815
B0x	2816	2817	2818	2819	2820	2821	2822	2823	2824	2825	2826	2827	2828	2829	2830	2831
B1x	2832	2833	2834	2835	2836	2837	2838	2839	2840	2841	2842	2843	2844	2845	2846	2847
B2x	2848	2849	2850	2851	2852	2853	2854	2855	2856	2857	2858	2859	2860	2861	2862	2863
B3x	2864	2865	2866	2867	2868	2869	2870	2871	2872	2873	2874	2875	2876	2877	2878	2879
B4x	2880	2881	2882	2883	2884	2885	2886	2887	2888	2889	2890	2891	2892	2893	2894	2895
B5x	2896	2897	2898	2899	2900	2901	2902	2903	2904	2905	2906	2907	2908	2909	2910	2911
B6x	2912	2913	2914	2915	2916	2917	2918	2919	2920	2921	2922	2923	2924	2925	2926	2927
B7x	2928	2929	2930	2931	2932	2933	2934	2935	2936	2937	2938	2939	2940	2941	2942	2943
B8x	2944	2945	2946	2947	2948	2949	2950	2951	2952	2953	2954	2955	2956	2957	2958	2959
B9x	2960	2961	2962	2963	2964	2965	2966	2967	2968	2969	2970	2971	2972	2973	2974	2975
BAx	2976	2977	2978	2979	2980	2981	2982	2983	2984	2985	2986	2987	2988	2989	2990	2991
BBx	2992	2993	2994	2995	2996	2997	2998	2999	3000	3001	3002	3003	3004	3005	3006	3007
BCx	3008	3009	3010	3011	3012	3013	3014	3015	3016	3017	3018	3019	3020	3021	3022	3023
BDx	3024	3025	3026	3027	3028	3029	3030	3031	3032	3033	3034	3035	3036	3037	3038	3039
BEx	3040	3041	3042	3043	3044	3045	3046	3047	3048	3049	3050	3051	3052	3053	3054	3055
BFx	3056	3057	3058	3059	3060	3061	3062	3063	3064	3065	3066	3067	3068	3069	3070	3071

Figure B-2 (Part 3 of 4). Hexadecimal/decimal conversion table

x =	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
C0x	3072	3073	3074	3075	3076	3077	3078	3079	3080	3081	3082	3083	3084	3085	3086	3087
C1x	3088	3089	3090	3091	3092	3093	3094	3095	3096	3097	3098	3099	3100	3101	3102	3103
C2x	3104	3105	3106	3107	3108	3109	3110	3111	3112	3113	3114	3115	3116	3117	3118	3119
C3x	3120	3121	3122	3123	3124	3125	3126	3127	3128	3129	3130	3131	3132	3133	3134	3135
C4x	3136	3137	3138	3139	3140	3141	3142	3143	3144	3145	3146	3147	3148	3149	3150	3151
C5x	3152	3153	3154	3155	3156	3157	3158	3159	3160	3161	3162	3163	3164	3165	3166	3167
C6x	3168	3169	3170	3171	3172	3173	3174	3175	3176	3177	3178	3179	3180	3181	3182	3183
C7x	3184	3185	3186	3187	3188	3189	3190	3191	3192	3193	3194	3195	3196	3197	3198	3199
C8x	3200	3201	3202	3203	3204	3205	3206	3207	3208	3209	3210	3211	3212	3213	3214	3215
C9x	3216	3217	3218	3219	3220	3221	3222	3223	3224	3225	3226	3227	3228	3229	3230	3231
CAx	3232	3233	3234	3235	3236	3237	3238	3239	3240	3241	3242	3243	3244	3245	3246	3247
CBx	3248	3249	3250	3251	3252	3253	3254	3255	3256	3257	3258	3259	3260	3261	3262	3263
CCx	3264	3265	3266	3267	3268	3269	3270	3271	3272	3273	3274	3275	3276	3277	3278	3279
CDx	3280	3281	3282	3283	3284	3285	3286	3287	3288	3289	3290	3291	3292	3293	3294	3295
CEx	3296	3297	3298	3299	3300	3301	3302	3303	3304	3305	3306	3307	3308	3309	3310	3311
CFx	3312	3313	3314	3315	3316	3317	3318	3319	3320	3321	3322	3323	3324	3325	3326	3327
D0x	3328	3329	3330	3331	3332	3333	3334	3335	3336	3337	3338	3339	3340	3341	3342	3343
D1x	3344	3345	3346	3347	3348	3349	3350	3351	3352	3353	3354	3355	3356	3357	3358	3359
D2x	3360	3361	3362	3363	3364	3365	3366	3367	3368	3369	3370	3371	3372	3373	3374	3375
D3x	3376	3377	3378	3379	3380	3381	3382	3383	3384	3385	3386	3387	3388	3389	3390	3391
D4x	3392	3393	3394	3395	3396	3397	3398	3399	3400	3401	3402	3403	3404	3405	3406	3407
D5x	3408	3409	3410	3411	3412	3413	3414	3415	3416	3417	3418	3419	3420	3421	3422	3423
D6x	3424	3425	3426	3427	3428	3429	3430	3431	3432	3433	3434	3435	3436	3437	3438	3439
D7x	3440	3441	3442	3443	3444	3445	3446	3447	3448	3449	3450	3451	3452	3453	3454	3455
D8x	3456	3457	3458	3459	3460	3461	3462	3463	3464	3465	3466	3467	3468	3469	3470	3471
D9x	3472	3473	3474	3475	3476	3477	3478	3479	3480	3481	3482	3483	3484	3485	3486	3487
DAx	3488	3489	3490	3491	3492	3493	3494	3495	3496	3497	3498	3499	3500	3501	3502	3503
DBx	3504	3505	3506	3507	3508	3509	3510	3511	3512	3513	3514	3515	3516	3517	3518	3519
DCx	3520	3521	3522	3523	3524	3525	3526	3527	3528	3529	3530	3531	3532	3533	3534	3535
DDx	3536	3537	3538	3539	3540	3541	3542	3543	3544	3545	3546	3547	3548	3549	3550	3551
DEx	3552	3553	3554	3555	3556	3557	3558	3559	3560	3561	3562	3563	3564	3565	3566	3567
DFx	3568	3569	3570	3571	3572	3573	3574	3575	3576	3577	3578	3579	3580	3581	3582	3583
E0x	3584	3585	3586	3587	3588	3589	3590	3591	3592	3593	3594	3595	3596	3597	3598	3599
E1x	3600	3601	3602	3603	3604	3605	3606	3607	3608	3609	3610	3611	3612	3613	3614	3615
E2x	3616	3617	3618	3619	3620	3621	3622	3623	3624	3625	3626	3627	3628	3629	3630	3631
E3x	3632	3633	3634	3635	3636	3637	3638	3639	3640	3641	3642	3643	3644	3645	3646	3647
E4x	3648	3649	3650	3651	3652	3653	3654	3655	3656	3657	3658	3659	3660	3661	3662	3663
E5x	3664	3665	3666	3667	3668	3669	3670	3671	3672	3673	3674	3675	3676	3677	3678	3679
E6x	3680	3681	3682	3683	3684	3685	3686	3687	3688	3689	3690	3691	3692	3693	3694	3695
E7x	3696	3697	3698	3699	3700	3701	3702	3703	3704	3705	3706	3707	3708	3709	3710	3711
E8x	3712	3713	3714	3715	3716	3717	3718	3719	3720	3721	3722	3723	3724	3725	3726	3727
E9x	3728	3729	3730	3731	3732	3733	3734	3735	3736	3737	3738	3739	3740	3741	3742	3743
EAx	3744	3745	3746	3747	3748	3749	3550	3751	3752	3753	3754	3755	3756	3757	3758	3759
EBx	3760	3761	3762	3763	3764	3765	3766	3767	3768	3769	3770	3771	3772	3773	3774	3775
ECx	3776	3777	3778	3779	3780	3781	3782	3783	3784	3785	3786	3787	3788	3789	3790	3791
EDx	3792	3793	3794	3795	3796	3797	3798	3799	3800	3801	3802	3803	3804	3805	3806	3807
EEx	3808	3809	3810	3811	3812	3813	3814	3815	3816	3817	3818	3819	3820	3821	3822	3823
EFx	3824	3825	3826	3827	3828	3829	3830	3831	3832	3833	3834	3835	3836	3837	3838	3839
F0x	3840	3841	3842	3843	3844	3845	3846	3847	3848	3849	3850	3851	3852	3853	3854	3855
F1x	3856	3857	3858	3859	3860	3861	3862	3863	3864	3865	3866	3867	3868	3869	3870	3871
F2x	3872	3873	3874	3875	3876	3877	3878	3879	3880	3881	3882	3883	3884	3885	3886	3887
F3x	3888	3889	3890	3891	3892	3893	3894	3895	3896	3897	3898	3899	3900	3901	3902	3903
F4x	3904	3905	3906	3907	3908	3909	3910	3911	3912	3913	3914	3915	3916	3917	3918	3919
F5x	3920	3921	3922	3923	3924	3925	3926	3927	3928	3929	3930	3931	3932	3933	3934	3935
F6x	3936	3937	3938	3939	3940	3941	3942	3943	3944	3945	3946	3947	3948	3949	3950	3951
F7x	3952	3953	3954	3955	3956	3957	3958	3959	3960	3961	3962	3963	3964	3965	3966	3967
F8x	3968	3969	3970	3971	3972	3973	3974	3975	3976	3977	3978	3979	3980	3981	3982	3983
F9x	3984	3985	3986	3987	3988	3989	3990	3991	3992	3993	3994	3995	3996	3997	3998	3999
FAx	4000	4001	4002	4003	4004	4005	4006	4007	4008	4009	4010	4011	4012	4013	4014	4015
FBx	4016	4017	4018	4019	4020	4021	4022	4023	4024	4025	4026	4027	4028	4029	4030	4031
FCx	4032	4033	4034	4035	4036	4037	4038	4039	4040	4041	4042	4043	4044	4045	4046	4047
FDx	4048	4049	4050	4051	4052	4053	4054	4055	4056	4057	4058	4059	4060	4061	4062	4063
FEx	4064	4065	4066	4067	4068	4069	4070	4071	4072	4073	4074	4075	4076	4077	4078	4079
FFx	4080	4081	4082	4083	4084	4085	4086	4087	4088	4089	4090	4091	4092	4093	4094	4095

Figure B-2 (Part 4 of 4). Hexadecimal/decimal conversion table

Appendix C. American National Standard Code for Information Interchange (ASCII)

Bit position	0-3	0000	0001	0010	0011	0100	0101	0110	0111
4-7	Hex	0	1	2	3	4	5	6	7
0000	0	NUL	DLE		0	@	P	`	p
0001	1	SOH	DC1	!	1	A	Q	a	q
0010	2	STX	DC2	"	2	B	R	b	r
0011	3	ETX	DC3	#	3	C	S	c	s
0100	4	EOT	DC4	$	4	D	T	d	t
0101	5	ENQ	NAK	%	5	E	U	e	u
0110	6	ACK	SYN	&	6	F	V	f	v
0111	7	BEL	ETB	'	7	G	W	g	w
1000	8	BS	CAN	(	8	H	X	h	x
1001	9	HT	EM	)	9	I	Y	i	y
1010	A	LF	SUB	*	:	J	Z	j	z
1011	B	VT	ESC	+	;	K	[	k	{
1100	C	FF	FS	,	<	L	\	l	|
1101	D	CR	GS	-	=	M	]	m	}
1110	E	SO	RS	.	>	N	^	n	~
1111	F	SI	US	/	?	O	_	o	DEL

Appendix D. Perforated Tape Transmission Code/Extended Binary Coded Decimal (PTTC/EBCD)

Bit Positions	0-3 (S,B,A,8)	0000	0001	0010	0011	0100	0101	0110	0111	1000	1001	1010	1011	1100	1101	1110	1111
4-7 (4,2,1,C)	Hex	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0000	0		8	@		Ⓝ NAK			h		*	¢					H
0001	1	Space			y		q	&		Space			Y		Q	+	
0010	2	1			z		r	a		=			Z		R	A	
0011	3		9	/		j			i		(	?		J			I
0100	4	2						b		<						B	
0101	5		O	s		k					)	S		K			
0110	6		#Ⓓ EOA	t		l			Ⓨ YAK			T		L			¬
0111	7	3			Ⓢ SOA		$	c		;			I		!	C	
1000	8	4						d		:						D	
1001	9			u		m						U		M			
1010	A			v		n			Horiz tab			V		N			Horiz tab
1011	B	5			LF see note 1		NL see note 2	e		%			LF see note 1		NL see note 2	E	
1100	C		Up-shift	w		o			Down-shift		Up-shift	W		O			Down-shift
1101	D	6			Ⓑ EOB		Back space	f		'			Ⓑ EOB		Back space	F	
1110	E	7					IDLE	g		>					IDLE	G	
1111	F		© EOT	x		p			DEL			X		P			DEL

Lower Case → | ← Upper Case

0	1	2	3	4	5	6	7

Bits

S	B	A	8	4	2	1	C

Terminal code structure

Start	B	A	8	4	2	1	C	Stop

Transmitted and received character

C is odd parity check bit for S, B, A, 8, 4, 2, 1.
On receiving operation, start and stop bits are deleted.
On transmitting operation, start and stop bits are added.

Shift (S) bit position 0 (lower case) or 1 (upper case) inserted on receive operations. Insertion/deletion is performed by equipment

Notes:
1. Line feed (LF) performs the indexing function
2. New line (NL) performs the carrier return and line feed function
3. Similar terms:
   downshift = lower case
   upshift = upper case

Appendix E. Priority List for Assembler Instructions

ICTL	must be the first statement of an assembly
TITLE EJECT PRINT SPACE COPY	any place
MACRO	before first CSECT
GBL	after macro prototype
LCL	follows globals
ACTR	follows locals
AIF AGO SETA SETB SETC ANOP MEXIT MNOTE	within macro definitions, anywhere between ACTR and MEND
MEND	must be last statement of a macro definition
ENTRY WXTRN EXTRN DSECT POP PUSH COM GLOBL ISEQ	anywhere after macro definitions
START	after macro definitions
CSECT	after macro definitions
EQU EQUR DC DS ORG USING DROP ALIGN PREF	if there is a START or CSECT statement, anywhere after it. If there is no START or CSECT, any of these instructions can begin a nameless CSECT.
END	must be last statement of an assembly

Appendix F. Summary of Constants

Type	*Implied length (bytes)*	*Length modifier range (bytes)*	*Specified by*	*Number of constants per operand*	*Exponent range*	*Scale range*	*Truncation/ padding side*	*Padding character*
C S P	as (2) needed	1–256 (1)	characters	one			right	blank
X	as (2) needed	1–256 (1)	hexadecimal digits	one			left	0
B	as (2) needed	1–256	binary digits	one			left	0
F	2	1–2	decimal digits	several	-85 to +75	–31 to +63	left	0
E	4	2–4	decimal digits	several	-85 to +75	0–14	right	0
A	2	1–4	any expression	several			left	0
N	2	2	name	several			assembled as X'0000'	
V	2	2	relocatable symbol	several			assembled as X'0000'	
W	2	2	relocatable symbol	several			assembled as X'0000'	
H	1	1	decimal digits	several	-85 to +75		left	0
D	4	1–4	decimal digits	several	-85 to +75	–31 to +63	left	0
L	8	2–8	decimal digits	several	-85 to +75	0–14	right	0

(1) In DS assembler instructions, C, S, P, and X type constants can have length specification to 65535.

(2) In DS assembler instruction types C, S, P, X, and B, the implied length is 1 when a length modifier and a constant value are not specified.

(3) Constants A, D, E, F, H, L are not truncated.

Appendix G. Macro Language Summary

The 4 figures in this appendix summarize the macro language described in Chapter 6 of this publication.

Figure G-1 is a summary of the expressions that may be used in macro instruction statements.

Figure G-2 indicates which macro language elements may be used in the name and operand entries of each statement.

Figure G-3 is a summary of the attributes that may be used in each expression.

Figure G-4 is a summary of the variable symbols that may be used in each expression.

Expression	*Arithmetic expressions*	*Character expressions*	*Logical expressions*
May contain	1. Self-defining terms 2. Count and number attributes 3. SETA and SETB symbols 4. SETC symbols whose value is 1-8 decimal digits 5. Symbolic parameters if the corresponding operand is a self-defining term 6. &SYSLIST (n) if the corresponding operand is a self-defining term 7. &SYSLIST (n,m) if the corresponding operand is a self-defining term 8. &SYSNDX 9. &SYSPARM whose value is 1-8 decimal digits	1. Any combination of characters enclosed in apostrophes 2. Any variable symbol enclosed in apostrophes 3. A concatenation of variable symbols and other characters enclosed in apostrophes 4. A request for a type attribute	1. SETB symbols 2. Arithmetic relations[1] 3. Character relations[2] 4. SETA expression[4]
Operators are	+, -, *, and / parentheses permitted	concatenation, with a period (.)	AND, OR, and NOT parentheses permitted
Range of values	-2^{31} to $+2^{31}-1$	0 through 127 characters	0 (false) or 1 (true)
May be used in	1. SETA operands 2. Arithmetic relations 3. Subscripted SET symbols 4. A subscript of &SYSLIST 5. Substring notation 6. Sublist notations 7. SETC operands 8. ACTR operands	1. SETC operands[3] 2. Character relations[2] 3. SETA operands (if 1-8 decimal digits)	1. SETB operands 2. AIF operands

[1] An arithmetic relation consists of two arithmetic expressions related by the operator GT, LT, EQ, NE, GE, or LE.

[2] A character relation consists of two character expressions related by the operator GT, LT, EQ, NE, GE, or LE. The type attribute notation and the substring notation may also be used in character relations. The maximum size of the character expressions that can be compared is 127 characters. If the two character expressions are of unequal size, then the smaller one will always compare less than the larger.

[3] Maximum of 64 characters will be assigned.

[4] The expression must be a valid SETA expression which resolves to 0 or 1.

Figure G-1. Expressions for conditional assembly

Statement	*Variable Symbols*										
	Global SET Symbols				*Local SET Symbols*			*System Variable Symbols*			
	Symbolic Parameter	*SETA*	*SETB*	*SETC*	*SETA*	*SETB*	*SETC*	*&SYSNDX*	*&SYSLIST*	*&SYSPARM*	*&SYSTIME &SYSDATE*
MACRO											
Prototype Statement	Name Operand										
GBLA		Operand									
GBLB			Operand								
GBLC				Operand							
LCLA					Operand						
LCLB						Operand					
LCLC							Operand				
Model Statement	Name Operation Operand	Name Operation Operand	Name Operation Operand	Name Operation Operand	Name Operation Operand	Name Operation Operand	Name Operation Operand	Name Operation Operand	Name Operation Operand	Name Operation Operand	Name Operand
SETA	Operand[2]	Name Operand	Operand[3]	Operand[7]	Name Operand	Operand[3]	Operand[7]	Operand	Operand[2]	Operand[7]	
SETB	Operand[4]	Operand[4]	Name Operand	Operand[4]	Operand[4]	Name Operand	Operand[4]	Operand[4]	Operand[4]	Operand[4]	
SETC	Operand	Operand[5]	Operand[6]	Name Operand	Operand[5]	Operand[6]	Name Operand	Operand	Operand	Operand	Operand
AIF	Operand[4]	Operand[4]	Operand	Operand[4]	Operand[4]	Operand	Operand[4]	Operand[4]	Operand[4]	Operand[4]	Operand[4]
AGO											
ACTR	Operand[2]	Operand	Operand[3]	Operand[2]	Operand	Operand[3]	Operand[2]	Operand	Operand[2]	Operand[2]	
ANOP											
MEXIT											
MNOTE	Operation	Operand	Operand	Operation	Operand	Operand	Operation	Operand	Operation	Operation	Operation
MEND											
Outer Macro		Name Operand	Name Operand	Name Operand	Name Operand	Name Operand	Name Operand			Name Operand	Name Operand
Inner Macro	Name Operand	Name Operand	Name Operand	Name Operand	Name Operand	Name Operand	Name Operand	Name Operand	Name Operand	Name Operand	Name Operand
Assembler Language Statement		Name Operation Operand	Name Operation Operand	Name Operation Operand	Name Operation Operand	Name Operation Operand	Name Operation Operand			Name Operation Operand	Name Operand

[1] Variable symbols in macro instructions are replaced by their values before processing.
[2] Only if value is self-defining term.
[3] Converted to arithmetic +1 or +0.
[4] Only in arithmetic or character relations.
[5] Converted to unsigned number.
[6] Converted to character 1 or 0.
[7] Only if one to eight decimal digits.

Figure G-2. (Part 1 of 2). Macro language elements

Statement	*Attributes*			*Sequence symbol*
	Type	*Count*	*Number*	
MACRO				
Prototype Statement				
GBLA				
GBLB				
GBLC				
LCLA				
LCLB				
LCLC				
Model Statement				Name
SETA		Operand	Operand	
SETB	Operand[1]	Operand[2]	Operand[2]	
SETC	Operand			
AIF	Operand[1]	Operand[2]	Operand[2]	Name Operand
AGO				Name Operand
ACTR		Operand	Operand	
ANOP				Name
MEXIT				Name
MNOTE				Name
MEND				Name
Outer Macro				Name
Inner Macro				Name
Assembler Language Statement				Name

[1] Only in character relations.
[2] Only in arithmetic relations.

Figure G-2. (Part 2 of 2). Macro language elements

Attribute	*Notation*	*May be used with:*	*May be used in:* *
Type	T'	Symbolic parameters, &SYSLIST (n), and &SYSLIST (n,m) inside macro definitions	1. SETC operand fields 2. Character relations
Count	K'	Symbolic parameters corresponding to macro instruction operands, &SYSLIST, and &SYSLIST (n) inside macro definitions	Arithmetic expressions
Number	N'	Symbolic parameters, &SYSLIST, and &SYSLIST (n) inside macro definitions	Arithmetic expressions
***Note.** There are definite restrictions in the use of these attributes. Refer to Chapter 6.			

Figure G-3. Attributes of macro-instruction operands

Variable symbol	*Defined by:*	*Initialized, or set to:*	*Value changed by:*	*May be used in:*
Symbolic[1] parameter	Prototype statement	Corresponding macro instruction operand	(Constant throughout definition)	1. Arithmetic expressions if operand is self-defining term 2. Character expressions
SETA	LCLA or GBLA instruction	0	SETA instruction	1. Arithmetic expressions 2. Character expressions
SETB	LCLB or GBLB instruction	0	SETB instruction	1. Arithmetic expressions 2. Character expressions 3. Logical expressions
SETC	LCLC ro GBLC instruction	Null character value	SETC instruction	1. Arithmetic expressions if value is self-defining term 2. Character expressions
&SYSNDX[1]	The assembler	Macro instruction index	(Constant throughout definition; unique for each macro instruction)	1. Arithmetic expressions 2. Character expressions
&SYSLIST[1]	The assembler	Not applicable	Not applicable	N' &SYSLIST in arithmetic expressions
&SYSLIST (n)[1] &SYSLIST (n,m)[1]	The assembler	Corresponding macro instruction operand	(Constant throughout definition)	1. Arithmetic expressions if operand is self-defining term 2. Character expressions
&SYSPARM[1]	You, in response to OPTIONS= prompt at the operator station	Value specified in a job control statement (null character value if not specified)	(Constant throughout assembly)	1. Arithmetic expressions if value is 1-8 decimal digits 2. Character expressions
&SYSDATE[1] &SYSTIME[1]	Supervisor at time assembly is started	Value obtained at beginning of assembly if timer support available (blanks if no timer support)	(Constant throughout assembly)	Character expressions

[1] All may only be used in macro definitions.

Figure G-4. Variable symbols

Appendix H. Assembler Language Summary

Name field	*Mnemonic*	*Operand field*	*Notes*
blank	ALIGN	{ WORD / ODD }	9
[label]	COM	blank	
blank	COPY	a symbol identifying a member of a partitioned data set	
[label]	CSECT	blank	
[label]	DC	[dup] type [mods] { 'value' / (value) } [,opnd2] ...	
blank	DROP	1–8 absolute register expressions, separated by commas	2
[label]	DS	[dup] type [mods] [{ 'value' / (value) }] [opnd2,] ...	
label	DSECT	blank	
blank	EJECT	blank	
blank	END	relocatable expression OR blank	3
blank	ENTRY	one or more relocatable symbols (entry symbols), separated by commas	
symbol	EQU	expression	4
symbol	EQUR	absolute expression	7
blank	EXTRN	one or more relocatable symbols (external symbols), separated by commas	
[label]	GLOBL	blank	
blank	ICTL	one to three decimal self-defining values of the form b, e, c	10
blank	ISEQ	two decimal self-defining values of the form l, r, or blank	11
blank	ORG	relocatable expression OR blank	
blank	POP	SECTION	
[label]	PREF	zero to four address specifications separated by commas	8
blank	PRINT	ON GEN DATA OFF NOGEN NODATA	
blank	PUSH	SECTION	
blank	SPACE	decimal value from 1–255 OR blank	5
[label]	START	self-defining term OR blank	6
id char	TITLE	character string up to 100 characters, enclosed in apostrophes	
blank	USING	addr, reg	1, 2
blank	WXTRN	one or more relocatable symbols (weak external symbols), separated by commas	

Notes.

1. Code any absolute or relocatable expression for the addr operand. Usually, you code either a single relocatable symbol or a self-defining term.

2. For a register operand, code any register expression that has a value of 0 through 7.

3. If you choose to code the operand in this instruction, you usually code a single relocatable symbol or a location counter reference.

4. Code any relocatable or absolute expression. Usually, you code a decimal or hexadecimal self-defining term, or a combination of a previously defined symbol and a self-defining term.

5. If you choose to code the operand in this instruction, you must code a decimal self-defining term.

6. If you choose to code the operand in this instruction, you usually code a hexadecumal self-defining term.

7. Code any absolute expression that has a value of 0 through 7.

8. An address specification may be omitted by coding two successive commas or by omitting trailing parameters.

9. WORD specifies that the location counter is to be reset if necessary to the next higher address which is evenly divisible by 2. ODD specifies that the location counter is to be reset if necessary to the next higher address which is not divisible by 2 (an odd byte boundary).

10.

Operand	Specifies	Allowable range
b	Begin column	1 through 40
e	End column	41 through 80
c	Continue column	2 through 40

11.

Operand	Specifies
l	leftmost column of field to be checked
r	rightmost column of field to be checked

Appendix J. Macro Language Instruction Summary

	Name field	*Mnemonic*	*Operand field*	*Notes*
(calling instruction) ——→	[label]	macro name	zero to 100 operands, separated by commas	
(prototype statement) ——→	[label]	macro name	zero to 100 symbolic parameters, separated by commas	
	blank	ACTR	any valid SETA expression	1
	sequence symbol or blank	AGO	sequence symbol	
	sequence symbol or blank	AIF	logical expression enclosed in parentheses, immediately followed by a sequence symbol with no intervening blanks	2
	sequence symbol	ANOP	blank	
	blank	GBLA	one or more variable symbols to be used as SET symbols, separated by commas	
	blank	GBLB	one or more variable symbols to be used as SET symbols, separated by commas	
	blank	GBLC	one or more variable symbols to be used as SET symbols, separated by commas	
	blank	LCLA	one or more variable symbols to be used as SET symbols, separated by commas	
	blank	LCLB	one or more variable symbols to be used as SET symbols, separated by commas	
	blank	LCLC	one or more variable symbols to be used as SET symbols, separated by commas	
	blank	MACRO	blank	
	[label]	MEND	blank	
	[label]	MEXIT	blank	
	[label]	MNOTE	message specification	
	symbol	SETA	arithmetic expression	1
	symbol	SETB	one of 3 options	
	symbol	SETC	one of 4 options	

Notes.

1. Normally, you code this operand as a decimal self-defining term.

2. Logical expressions contain combinations of variable symbols, logical and relational operators, and arithmetic and character expressions. Normally, you code this operand in the form:
   (variable-symbol relational-operator self-defining term) or
   (variable-symbol relational-operator 'character-string').

Index

Program Preparation Subsystem
Macro Assembler: User's Guide
SC34-0124-0

READER'S COMMENT FORM

YOUR COMMENTS, PLEASE . . .

Your comments assist us in improving the usefulness of our publications; they are an important part of the input used in preparing updates to the publications. All comments and suggestions become the property of IBM.

Please do not use this form for technical questions about the system or for requests for additional publications; this only delays the response. Instead, direct your inquiries or requests to your IBM representative or to the IBM branch office serving your locality.

Corrections or clarifications needed:

Page Comment

Cut or Fold Along Line

What is your occupation?_____

Number of latest Technical Newsletter (if any) concerning this publication: _____

Please indicate your name and address in the space below if you wish a reply.

_____

_____

_____

Thank you for your cooperation. No postage stamp necessary if mailed in the U.S.A.
(Elsewhere, an IBM office or representative will be happy to forward your comments.)

Your comments, please . . .

This manual is part of a library that serves as a reference source for IBM systems. Your comments on the other side of this form will be carefully reviewed by the persons responsible for writing and publishing this material. All comments and suggestions become the property of IBM.

Fold Fold

First Class
Permit 40
Armonk
New York

Business Reply Mail
No postage stamp necessary if mailed in the U.S.A.

IBM Corporation
Systems Publications, Dept 27T
P.O. Box 1328
Boca Raton, Florida 33432

Fold Fold

International Business Machines Corporation
General Systems Division
5775D Glenridge Drive N.E.
P.O. Box 2150, Atlanta, Georgia 30301
(U.S.A. only)

Cut Along Line

IBM Series/1 Program Preparation Subsystem Macro Assembler: User's Guide Printed in U.S.A. SC34-0124-0

International Business Machines Corporation

General Systems Division
5775D Glenridge Drive N.E.
P. O. Box 2150
Atlanta, Georgia 30301
(U.S.A. only)

IBM Series/1 Program Preparation Subsystem: Macro Assembler User's Guide Printed in U.S.A. SC34-0124-0

SC34-0124-0

THE COMPUTER MUSEUM HISTORY CENTER
1 026 2099 8