SIGNETICS MICRO ASSEMBLER REFERENCE MANUAL

$4.00

SIGNETICS MICRO ASSEMBLER REFERENCE MANUAL

PREFACE

The Signetics Micro Assembler is a FORTRAN program that has been developed as a design tool for writing microprograms. The Micro Assembler frees the microprogrammer from the tedium of keeping track of individual bits within the micro-instruction as the microprogram is being written. Instead of hand coding binary digits, the Micro Assembler allows the microprogrammer to define his own assembler, tailored to the specific needs of his microprogrammed system. Once the microprogrammer has defined his own assembly language, the microprogram can be written in that language. The Micro Assembler then assembles the microprogram and generates paper tapes that can be used to program PROMs or load RAMs. In addition to producing PROM programming tapes, the Micro Assembler also generates listings that can serve as the micro-program's primary documentation.

The Micro Assembler has been developed to support Signetics' expanding bipolar microprocessor product line. Devices directly supported are the:

- 8X02 control store sequencer
- N3002 two-bit RALU (Register Arithmetic Logic Unit) slice
- N2901-1 four bit RALU slice

Data sheets on the above products are available from Signetics.

Although specifically intended for use with Signetics' bipolar microprocessor products, the flexibility of the Micro Assembler enables it to handle virtually all microprogrammed applications.

SIGNETICS MICRO ASSEMBLER REFERENCE MANUAL

TABLE OF CONTENTS

TABLE OF CONTENTS (Continued)

PART FIVE – THE MICRO FORMAT PROGRAM

APPENDICES

PART ONE – INTRODUCTION TO THE MICRO ASSEMBLER

MICROPROGRAMMING

Since its introduction by Wilkes in 1951[1], microprogramming has been praised in the literature as a powerful alternative to random logic CPU design. The basic concept, as outlined by Wilkes, is to have the instruction (which has been fetched from program storage) address a micro control memory. This memory, called Micro Control Store, then outputs a control word (the microinstruction) which determines the operation of each functional block of the CPU. This control memory approach contrasts with the rather ad hoc approach of random logic instruction decoding, where each CPU control signal is the result of a complex logical equation implemented with simple SSI (Small Scale Integration) gates.

Placing the CPU's control logic in memory (usually firmware: PROMs or ROMs), provides many advantages over the random logic approach. The entire design process benefits from the orderly structure of the Micro Control Store instruction decode. Each control field is generated independently of the other control fields. Hence, the various functional blocks can be developed separately and need not be combined until the final phase of system development. The microprogrammed system's final configuration is also much more flexible than its random logic counterpart. Since the control logic is essentially stored in memory, radical changes in system operation may be implemented by changing just a few words of the microprogram (residing in Micro Control Store). Likewise, a system's overall performance can easily be enhanced by adding a few bits to the Micro Control Store's output (that is, by adding a new control field to the microinstruction) with minimum impact on the existing hardware. These features simplify both the design and production phases of a microprogrammed system's development cycle.

With all of its advantages, microprogramming would be used extensively were it not for two serious drawbacks. The first is the cost of the Micro Control Store memory. Since system throughput is directly related to system speed, most applications require that the Micro Control Store be extremely fast. This requirement, when coupled with the fact that a microprogram can be fairly large (a medium size CPU requiring a 512 microinstruction by 48 control bit Micro Control Store memory) has restricted microprogrammed applications to systems that could absorb the high cost of fast Micro Control Store memory. However, this situation is changing rapidly. The cost of high speed ROMs and PROMs is dropping at an ever increasing rate. As the price of solid state memory continues to drop, microprogrammed techniques will find increasing use not only in CPU designs but in sequential state machines as well.

The second drawback is the availability of software support for writing microprograms. Historically, only the large main-frame computer manufacturers could afford the development costs associated with developing a special purpose micro assembler. But this situation has also changed. With its new Micro Assembler, Signetics provides the small manufacturer with a cost effective microprogram development tool. The Micro Assembler is a Fortran program available either from the time-sharing services or as a tape that can be run directly on the user's computing facility. The Micro Assembler allows the microprogrammer to first define his own specialized assembler, and then assemble his microprogram into PROM programming paper tapes. These paper tapes can then be used to program Micro Control Store PROMs or load Control Store RAM (in the case of writable Micro Control Store).

THE POWER OF MNEMONICS

Simply stated, a microprogram is a collection of microinstructions stored in some form of memory (usually ROM or PROM). Each address of this memory or Micro Control Store represents a machine state as determined by the microinstruction stored at that address. The microinstruction is a group of control fields, each field controlling some aspect of the machine's operation. For example, one control field might define the logical or arithmetic operation performed by the system ALU (Arithmetic Logic Unit); another control field might determine the source of the ALU's operands; and yet another the destination of the ALU results.

The output of the Micro Control Store is a collection of binary control bits. Therefore, the state of each control field can be expressed as a binary value, and in this manner each microinstruction can be formed. Using this technique, the microprogrammer would lay out a large sheet of paper and list the address of each microinstruction in a column. Then, each control field is assigned a binary value at each microinstruction address. This method is feasible and has been used extensively in the past. However, assuming even a relatively small microprogram consisting of 128 microinstructions, each 24 bits wide, the programming sheet would appear as an array of 3,968 binary digits. Clearly, this format is cumbersome at best and very confusing at worst.

The task of writing a microprogram would be far easier if the microprogrammer could use an assembler. An assembler would allow each control field to be given a symbolic name.

[1] Wilkes, M.V., "The Best Way To Design An Automatic Calculating Machine," Manchester U. Computer Inaugural Conference, Pg. 16, 1951.

Likewise, each individual control field state could be assigned a symbolic name that would reflect its function. With this approach, each microinstruction could be written as a sequence of micro-function mnemonics. With this format, the microinstruction would read almost like a sentence, its operation easily analyzed by reading the symbolic function of each control field. Instead of,

Address	Function	SourceA	SourceB	Destination
010 0011	00110	1110	0110	10101
010 0100	11100	0001	1100	00000

we would have something like,

```
MULT     SHIFTR   (REG2,   NOP,    REG2)   "BEGIN MULTIPLY"
MULT+1   ADD      (REG2,   REG5,   REG2)
```

There is no question that an assembler would significantly decrease the amount of energy required to both compose and read a microprogram.

Between descending PROM prices and the availability of a cheap microprogram assembler, microprogramming should become a cost effective solution to many logic design problems that have been the exclusive domain of SSI and MSI (Medium Scale Integration) TTL logic.

THE SIGNETICS MICRO ASSEMBLER

The Signetics Micro Assembler operates just like a conventional assembler in that it reads source statements written in a symbolic assembly language, and processes the statements to produce a machine language representation for each microinstruction. The machine language microinstruction is output as a binary word that can be stored in the Micro Control Store memory.

The major difference between the Micro Assembler and other assemblers is that the Micro Assembler allows the microprogrammer to design his own special purpose assembly language before he begins writing the microprogram. Assembler features that can be defined by the microprogrammer are listed in Table 1.1.

TABLE 1.1

USER DEFINABLE FEATURES OF THE MICRO ASSEMBLER

- The number of bits in the microinstruction (Micro Control Store width).
- The number of microinstructions in the microprogram (Micro Control Store length).
- Control Field placement within the microinstruction.
- The mnemonic name and size (in bits) of each control field.
- The mnemonic name for each control field value (generally suggestive of the field value's function).
- The organization of Micro Control Store as physical PROM modules (e.g., six 512x8 PROMs for 512x48 microprogram).
- The mapping of the assembled microprogram into Micro Control Store PROM modules.
- The format of the PROM programming tapes punched at the end of the assembly and PROM assignment processes.

The Micro Assembler consists of two separate programs. These are the Micro Assembly Program and the Micro Format Program. The Micro Assembly Program assembles the user source representation of the microprogram and generates an intermediate object text. This intermediate object text is then input to the Micro Format Program. The Micro Format Program partitions the machine language microprogram into PROM modules. Once the microprogram has been assigned to Control Store PROM modules, the Micro Format Program punches paper tapes that can be used to program the Micro Control Store PROMs.

Both the Micro Assembly Program and the Micro Format Program are subdivided into two sections. The first section in each case is a command file that defines how each program will operate. For the Micro Assembly Program, the first section is called the Microinstruction Definition Section. This section allows the user to define the exact nature of his assembler (as outlined in Table 1.1). The second section of the Micro Assembly Program's input is the user's microprogram assembly source, written in the assembly language defined by the Microinstruction Definition Section.

The first section input to the Micro Format Program is called the Format Command File. In the Format Command File, the user specifies how the microprogram will be partitioned in Micro Control Store PROM modules. The Format Command File also allows the microprogrammer to specify the format of the PROM programming tapes output by the Micro Format Program. The second section input to the Micro Format Program is the intermediate object text generated by the Micro Assembly Program.

BASIC STATEMENTS OF THE MICRO ASSEMBLER

User input to the Micro Assembler is divided into three of the four sections previously mentioned: the Microinstruction Definition Section, the Program Section and the Format Command File. Each section is written as a series of statements to the Micro Assembler. With each statement, the microprogrammer requests some action to be performed by the Micro Assembler (e.g., name and assign a control field to the microinstruction, assemble a microinstruction, etc.). What follows is a summary of the basic statements with which the microprogrammer communicates to the Micro Assembler. The statements appear under the input section where they may be used.

Microinstruction Definition Section Statements (Input to Micro Assembly Program)

1. The INSTRUCTION Statement
   The INSTRUCTION Statement specifies the horizontal width of the Micro Control Store (i.e., the number of bits in the microinstruction).
2. The FIELD Statement
   The FIELD Statement names a control field and assigns it a location in the microinstruction.
3. The MICROP Statement
   The MICROP Statement defines a Microp for the Micro Assembler. A Microp is a symbol that represents the value assigned to one or more control fields. The MICROP Statement assigns a symbolic name to a microp and defines field assignments that will be made when the microp is encountered in a Microinstruction Statement.
4. The INTRINSIC Statement
   The INTRINSIC Statement includes predefined microps in the user's Microinstruction Definition Section. Intrinsic microps supporting Signetics LSI microprocessor elements N3002, 2901 and 8X02 are available.

Program Section Statements (Input to Micro Assembly Program)

1. The PROGRAM Statement
   The PROGRAM Statement defines the physical size of the Micro Control Store where the microprogram is to reside. It specifies the width and length of the Micro Control Store (i.e., the number of bits in the microinstruction and the number of microinstructions respectively).
2. The Microinstruction Statement
   The Microinstruction Statement assigns values to each control field of the microinstruction. The microprogram is input to the Micro Assembler as a sequence of Microinstruction Statements.
3. The DCL (Declare) Statement
   The DCL Statement allows the microprogrammer to specify literal data values as microinstructions. This statement is useful for building PROM look-up tables and other auxiliary PROMs that might be included in a microprogrammed application.

Format Command File Statements (Input to the Micro Format Program)

The Format Command File partitions the microprogram into PROM modules and produces a PROM programming tape for each PROM of the Micro Control Store memory. Each PROM requires four statements to generate its programming tape. These are the FORMAT Statement, the MEMORY Statement, the OUTPUT Statement and the SELECT Statement.

1. The FORMAT Statement
   The FORMAT Statement specifies the paper tape format of the PROM programming tapes generated by the Micro Format Program. The FORMAT Statement's format remains in effect until the Micro Format Program encounters another FORMAT Statement. Hence, a FORMAT statement must precede the definition of all PROM modules with its format.

2. The MEMORY Statement
   The MEMORY Statement specifies the bit dimensions of the PROM module (e.g., the Signetics 82S115 is 512x8).
3. The OUTPUT Statement
   The OUTPUT Statement directs the Micro Format Program to output a section of the microprogram (coded as the intermediate object text) into the PROM module specified by the preceding MEMORY Statement.
4. The SELECT Statement
   The SELECT Statement assigns specific microinstruction bits to a PROM module. It allows arbitrary placement for PCB (Printed Circuit Board) layout considerations.

Micro Assembler Directives (Input to the Micro Assembly Program).

The Micro Assembler Directives are a group of statements that allow the microprogrammer to define the value of symbols, force the program counter to a non-sequential value and control the generation of the program listings. The ORG statement may only be used in the Program Section. All other directives may be freely interspersed throughout both the Microinstruction Definition Section and the Program Section.

1. The EQU (Equate) Statement
   The EQU Statement assigns a value to a mnemonic name (symbol).
2. The SET Statement
   The SET Statement, like the EQU Statement, assigns a numeric value to a symbol. However, unlike the EQU Statement, Symbols defined with the SET Statement may be later redefined with another SET Statement.
3. The ORG (Origin) Statement
   The ORG Statement relocates the Micro Assembly Program's location counter.
4. The LIST Statement
   The LIST Statement specifies which listings will be produced.
5. The OBJECT Statement
   The OBJECT Statement controls the generation of the intermediate object text as an output listing.
6. The SPACE Statement
   The SPACE Statement produces blank lines in the output listings.
7. The EJECT Statement
   The EJECT Statement terminates the listing output on any given listing page.

DESIGNING A MICROPROGRAM WITH THE MICRO ASSEMBLER

Designing a microprogram begins by defining the hardware environment that will execute the microprogram. The hardware will naturally group into functional blocks (e.g., the ALU section, the next address section, I/O structures), that will require a control field to determine their function during any given microcycle. The microinstruction word is then assembled as a collection of the necessary control fields. A typical microinstruction is presented in Figure 1.1.

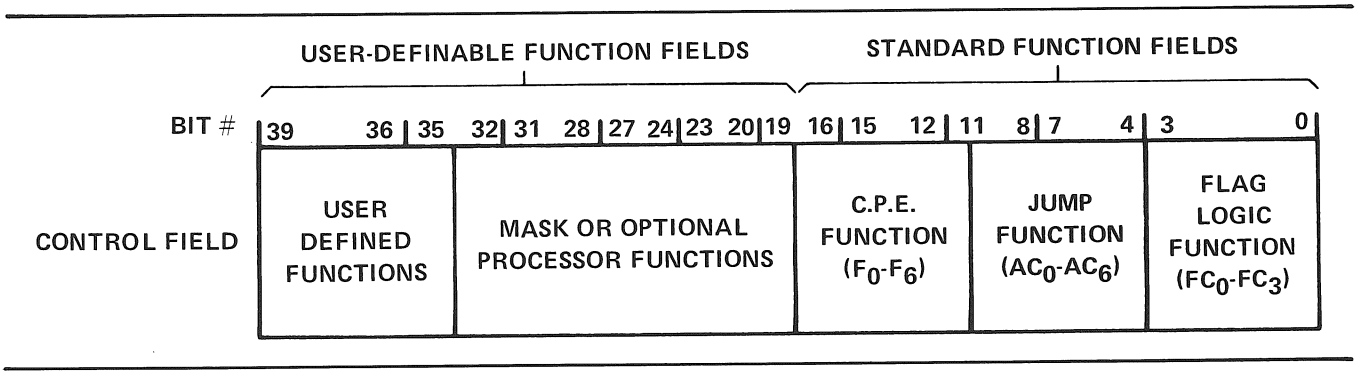


FIGURE 1.1

Once the hardware architecture and its resulting microinstruction have been defined, the microprogrammer may begin to prepare his input to the Micro Assembler. The microprogram object (PROM programming tapes) is produced in three steps. These steps are listed below:

1. Define the Assembly Language: The Microinstruction Definition Section. This section defines all of the symbols that will be used to write the microprogram assembly source. This includes naming all of the control fields and defining the microps that will make particular value assignments to the control fields.
2. Write the Microprogram: The Program Section. The microprogram is written in the assembly language defined in the Microinstruction Definition Section. The microprogram consists of a sequence of Microinstruction Statements. Each Microinstruction Statement specifies the state of each control field in the microinstruction by

referencing microps defined in the Microinstruction Definition Section. The Microinstruction Definition Section and the Program Section are input to the Micro Assembly Program.

3. Assign the Assembled Microprogram to PROMs: The Format Command File. The Micro Format Program accepts as input the assembled microprogram from the Micro Assembly Program (output as the intermediate object text) and the Format Command File written by the microprogrammer. By interpreting the Format Command File, the Micro Format Program partitions the microprogram (which has been assembled into a binary array) into PROM modules. The Micro Format Program generates a PROM programming tape for each PROM module. The format of the PROM programming tapes can also be specified in the Format Command File.

Figure 1.2 presents a diagram of the interaction between the microprogrammer and the Micro Assembler in the process of assembling a microprogram and generating PROM programming tapes for the Micro Control Store PROMs.

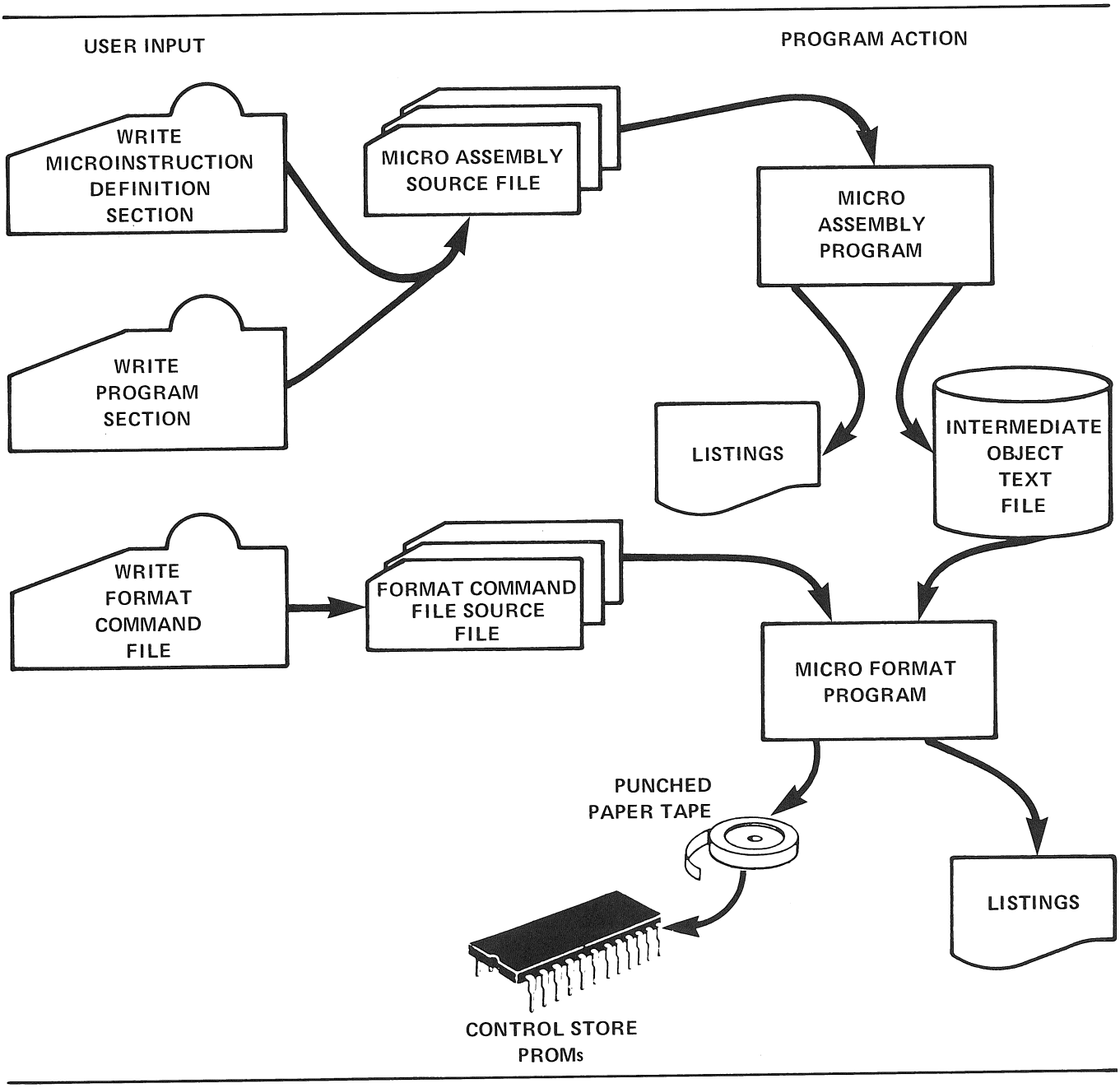


FIGURE 1.2

PART TWO – THE LANGUAGE

LANGUAGE ELEMENTS

The microprogrammer communicates with the Micro Assembler by writing a sequence of logical statements which the Micro Assembler can interpret in a meaningful way. These statements (the source input) are formed by combining language elements according to certain rules. This section will present the language elements. The remainder of the Reference Manual will be devoted to the rules concerning their use.

Characters

The basic atom of the Micro Assembler language is a character. The supported characters are listed in Table 2.1.

TABLE 2.1

Alphabetic Characters:	A through Z
Numeric Characters:	0 through 9
Special Characters:	Blank
	" Double Quote
	$ Dollar Sign
	' Quote
	(Left Parenthesis
	) Right Parenthesis
	+ Plus Sign
	, Comma
	– Minus Sign
	: Colon
	; Semicolon
	= Equal Sign
	@ At Sign
	_ Underline

The remaining ASCII characters may only be used in quoted strings of characters (character constants or in comments. Actually the character set available to the user may be limited by the FORTRAN I/O facility of the host machine where the Micro Assembler resides.

Symbols

A symbol is a group of characters that is used as a name. Typical symbols are:

- Mnemonic names for numerical values
- Symbolic addresses
- Mnemonic names for microinstruction fields
- Reserved words (Statement Op-Codes, Keywords and Expression Operators)

Rules governing the construction of symbols:

1. Symbols may consist of the alphabetic characters, the numeric characters and the "at sign" (@).
2. Symbols can be from 1 to 28 characters long.
3. The first character of a symbol must be an alphabetic character or the "at sign."
4. Symbols must be contained entirely on one line of source (i.e., symbols may not be split between lines).
5. Blanks may not be embedded within a symbol.

The special character, underline (_), may be embedded within a symbol to improve readability. The underline does not affect the meaning of the symbol and is removed when the Micro Assembler collects the symbol (it is, however, reproduced on the source listing). Underlines may not begin or end a symbol.

Table 2.2 lists examples of both valid and invalid symbols.

TABLE 2.2

VALID SYMBOLS	
A	
MOVE	
LONG_SYMBOL_NAME	(= LONGSYMBOLNAME)
VERY@LONG@SYMBOL@NAME	
FIELD7	
@45	
INVALID SYMBOLS	**Reason**
2Y	First character is not an alphabetic character or @.
_ADDR	First character is underline.
AB CD	Embedded blank.
DATA/LIST	Embedded delimiter (illegal character)
TOO@MANY@CHARACTERS@IN@SYMBOL	More than 28 characters.

Self-Defining Constants

A self-defining constant is a group of characters that specifies a constant value. The value of a self-defining constant is implicit in its representation. The Micro Assembler supports two types of self-defining constants: numeric constants and ASCII character constants.

Numeric Constants. Numeric constants are used to assign a numerical value to a symbolic name (e.g., when specifying a Field value for a Microp), as terms of an expression or anywhere a numerical value is appropriate. Numeric constants have the following format:

nnnnR

Where:

n is a numeric character,

and

R is the radix of the numeric constant.

The radix, R, is an alphabetic character that specifies the base of the numeric constant, and hence, the valid characters for "nnnn." The Micro Assembler recognizes the radixes presented in Table 2.3.

TABLE 2.3

Radix	Valid Characters for "nnnn"
B — Binary	0 and 1
O or Q — Octal	0 through 7
D — Decimal	0 through 9
H — Hexadecimal	0 through 9, A through F (A through F represent decimal 10 through 15, respectively)

If "R" is omitted from a numerical constant, the radix is assumed to be D (decimal).

Rules governing the construction of numeric constants:

1. The maximum number of characters is set with the "M toggle." (Various Micro Assembler parameters are set by special assembler commands called toggles — see Appendix A). If the M toggle is not set by the user, the default value for maximum number of bits in a numeric constant is 128 bits.
2. The first character of a numeric constant must be a numeric character. (For hexadecimal constants this requirement can be met by adding a leading zero. The Micro Assembler ignores leading zeros.)
3. A numeric constant must be contained entirely on one line of source.
4. A blank may not be embedded within a numeric constant.

As with symbols, the underline may be embedded within a numeric constant to improve readability. The underline is reproduced on the source listing but is ignored during the assembly process. Numeric constants may not begin or end with an underline.

Table 2.4 lists examples of valid and invalid numeric constants.

TABLE 2.4

VALID NUMERIC CONSTANTS	
13 1101B 15Q 001_101B 13D 0DH	Each example has a decimal value of 13.
INVALID NUMERIC CONSTANTS	**Reason**
12B	2 is an invalid character for the binary radix.
12 34Q	Embedded blank.
19.95	Embedded delimiter (illegal character).
FB5AH	First character is not numeric.
3F25DA70FH	Exceeds maximum word size (assuming maximum word size of 32 bits set by M toggle).

ASCII Character Constants. Occasionally, it is necessary to store the binary form of an ASCII character in memory. An example of this requirement might be the storage of human interface messages in PROM. So that the microprogram designer need not burden himself with the tedious task of converting ASCII characters into their binary code, the Micro Assembler has provisions for accepting ASCII character strings in the form of self-defining constants. When the Micro Assembler detects a character constant string, it converts each character into an 8-bit ASCII byte (7-bit

ASCII with high-order zero appended). This conversion is independent of the internal FORTRAN character representation of the host machine.

Rules governing the construction of character constants:

1. The maximum number of characters is determined by the value of the M toggle. The default value is 128 bits (16 characters).
2. Character constants must begin and end with a single quote (apostrophe). Within the single quotes, any supported ASCII character may be used. (A single quote within the character constant may be specified by using two adjacent single quotes. The Micro Assembler will discard one of the two.)
3. A character constant must be entirely contained on one line of source.

The following list shows some representative character constants and their equivalent numerical values.

'A' = 41H

'7' = 37H

'$' = 24H

'123' = 313233H

'DON' 'T' = 444F4E2754H

Table 2.5 lists examples of valid and invalid ASCII character constants.

TABLE 2.5

VALID CHARACTER CONSTANTS	
'*'	
'ABCDEFG'	
'FILE NAME'	
' '	(blanks)
'MARY' 'S PROGRAM'	(=MARY'S PROGRAM)
'$2.49'	
INVALID CHARACTER CONSTANTS	**Reason**
DON' 'T DO THIS'	Does not begin with single quote.
'DON' 'T DO THIS EITHER	Doesn't end with single quote.
'0123456789ABCDEFG'	Too many characters (assuming maximum word size of 128 bits).

Expressions

Self-defining constants and symbolic names that represent numerical values may be combined with operators to compute values. These combinations are called expressions. An expression may be used anywhere a numerical value is required in the user's source.

The expression operators are the special characters: + (add) and – (subtract). Subtraction is performed by two's complement addition.

Expressions are evaluated left to right. Should the microprogram designer wish to implement a more complex expression format, he may use parenthesis to group subexpressions. The following illustrates expression calculation priority:

Expression	Order of Calculation
A+B–C+D	A+B, (A+B)–C, ((A+B)–C)+D
(A+B) – (C+D)	A+B, C+D, (A+B) – (C+D)

Rules governing the construction of expressions:

1. Calculations are performed at the maximum word size of the Micro Assembler (set by the M toggle – default 128 bits). Carry out of the most significant bit is ignored.
2. The expression operators must be between two operands. (An operand is a self-defining constant, a symbol that can be resolved to a numerical value, or a parenthesized sub-expression.)
3. The subtract operator (–) may be used with a single operand at the beginning of an expression or subexpression. In this instance, the first character of the expression must be the subtract operator (–).
4. A single operand expression without an operator is valid.
5. Expressions *may* be split between two or more lines of source.
6. Blanks and comments may be placed between operands and operators within an expression.

Within the Microprogram Section of the user's source, the current value of the location counter may be used as an operand in an expression. The location counter points to the microprogram address that will be assigned to the current microinstruction being written. The location counter is referenced with the special character "dollar sign" ($).

Table 2.6 lists some examples of both valid and invalid expressions.

TABLE 2.6

VALID EXPRESSIONS	
A+B+1	
A + B + 1	
END – (OVER – 8)	
– DISPLACEMENT	
TABLE + 'A'	(Character constant 'A' = 65D)
$ + 0A3H	
BUF + (–5DH)	
INVALID EXPRESSIONS	**Reason**
Q + + 9	Operand is missing between the two add operators.
BUS + – 6	Subtract operator with single operand only valid at beginning of expression or sub-expression.
X + Y (Z)	Last two operands are not separated by an operator.

LANGUAGE STATEMENTS

In the preceding section, the basic language elements of the Micro Assembler were described. These language elements are combined by the microprogram designer to form statements which can be interpreted by the Micro Assembler.

Each statement is a command to the Micro Assembler. The Micro Assembler reads a statement, interprets its meaning, and performs the specific action requested by the statement.

The source input to the Micro Assembler is a sequence of statements. The ultimate purpose of the source statements is to direct the Micro Assembler in the production of the various program listings (both source and object) and in the generation of intermediate output object code, which, when input to the Micro Format program, is translated into output suitable for loading into the user's Control Store memory.

The source input to the Micro Assembler is organized into lines of 80 characters. All 80 character positions may be used for source text. The Micro Assembler processes the user's source without regard to line boundaries. Hence, a single statement may utilize several adjacent lines. Also, blanks may be inserted anywhere within statements to improve readability and to accommodate source line boundaries. (The only exception to the above are symbols and self-defining constants which may not cross source line boundaries nor contain embedded blanks.)

A statement is terminated by a semicolon (;). The first statement of the user's source program begins at the first character of the first source line and is terminated by a semicolon. Thereafter, all succeeding statements begin after the semicolon terminating the previous statement. Properly terminated, multiple statements are allowed on one line of user source. A null statement consists of zero or more blanks followed by a semicolon. (The null statement performs no action.)

Each statement is divided into three logical segments:

a. statement label (optional except for EQU and SET statements)

b. statement body

c. comment (optional)

A thorough description of the three statement segments is provided in the following sections.

Statement Labels

Excepting the SET and EQU statements, a statement label is used to give a mnemonic name to a microinstruction's address. (For the SET and EQU statements, the statement label is used as the symbolic name to which a numerical value will be assigned.)

As a symbolic address, a statement label is assigned the current value of the location counter. This feature allows the microprogrammer to reference a labeled microinstruction elsewhere in his microprogram.

The statement label (when present) is the first segment of a statement. It is terminated by a colon (:). Multiple statement labels are allowed provided each is followed by a colon.

Examples of statement labels:

```
        BEGIN: STATEMENT
               BODY
               "COMMENT" ;
BRANCH:  JUMP: STATEMENT
               BODY
               "COMMENT" ;
        ZERO: EQU 0B ;
        STAR: EQU '*' ;
```

The BEGIN and JUMP statements: statement label given value of location counter.

ZERO: ZERO given value of binary zero.

STAR: STAR given value of decimal 42.

The Statement Body

The statement body contains the information that requests a specific action to be performed by the Micro Assembler. The first language element the Micro Assembler finds in a statement body must be a symbol. The symbol must be a statement Op-Code such as INSTRUCTION or EQU, or it must be a mnemonic name for a microinstruction Field or Microp. After the first symbol, the format of the remainder of the statement body is determined by the type of statement being written. Examples of valid statements will be given in the description of each statement.

Comments

Every statement may contain a comment to help document the user's source program. A comment is reproduced in the source listing but is not processed during the assembly.

Comments must begin and end with a double quote ("). Any supported character except the double quote may be used inside a comment. Several lines of source may be used for a single comment. Although generally placed at the end of a statement, a comment may be placed anywhere within a statement where a blank is valid.

Statement Description Format

A good deal of the rest of this manual will be concerned with presenting and describing the Micro Assembler's statements. To ease the task of describing the various statements and provide a logical framework from which the particular aspects of each statement may be discussed, the following statement description format will be adapted.

Statement Description Format

[label:] . . . OP-CODE { name <u>s</u> } { KEYWORD= s / c / e } [name e] . . . ["COMMENT"] ;

Where:

OP-CODE	The Op-Code will always appear unenclosed and capitalized.
{ }	Braces indicate an operand that is required by the Op-Code.
[ ]	Brackets indicate an operand or some other part of the statement that is optional.
KEYWORD	Keywords will always be enclosed and capitalized. A keyword is a reserved word that indicates to the Micro Assembler that a specific parameter is to follow.
name	The name of the operand enclosed by braces or brackets is given in lower case letters.
name e	Following the name of an operand is a small letter designating the language element that may be used for the operand.
s c e	Stacked elements indicate that a selection is possible. Any element may be chosen.
<u>s</u>	An underline means the element must have been previously defined in the user's source.
. . .	Three dots following an operand indicate that the operand may be repeated any number of times.
label: KEYWORD=	Where required, delimiters will be shown.
Language Element Variables:	s — symbol c — self-defining constant e — expression

FIGURE 2.1

SPECIFYING A CUSTOM ASSEMBLER – THE MICROINSTRUCTION DEFINITION SECTION

Once the microprogram designer has decided on the format of his microinstruction, he is ready for the first phase of his microprogram development cycle: writing the Microinstruction Definition Section. The Microinstruction Definition Section lays the groundwork for the microprogram to follow. In this section, the microprogram designer communicates to the Micro Assembler all of the pertinent information about the microinstruction that will be necessary to assemble the microprogram. This information includes:

- The width (in bits) of the microinstruction.
- The mnemonic name for each Field within the microinstruction.
- The width (in bits) and position of each Field.
- Mnemonic names for specific values of Fields (Microps).
- Default values for the Fields.
- Request of Intrinsic Microps available with the Micro Assembler.

The remainder of this chapter provides a detailed description of how to communicate this information to the Micro Assembler.

Microinstruction Definition

The definition of a microinstruction is accomplished with a sequence of statements that specify the microinstruction's width, and stipulate the name, width and location of each Field within the microinstruction. The microinstruction definition begins with an INSTRUCTION statement. The INSTRUCTION statement is followed by FIELD statements. The microinstruction definition is terminated by an END INSTRUCTION statement. Listing directives may be interspersed within the microinstruction definition.

The INSTRUCTION Statement

The INSTRUCTION statement specifies the number of bits in the microinstruction. The INSTRUCTION Statement has the format illustrated in Figure 2.2.

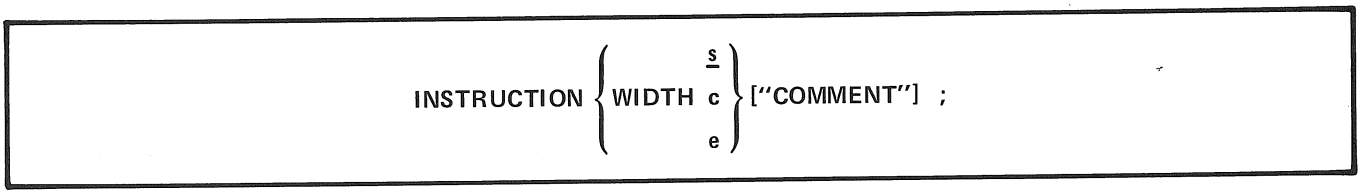


FIGURE 2.2

This statement begins with the INSTRUCTION Op-Code. The INSTRUCTION Op-Code is followed by the width operand which consists of the keyword WIDTH and its argument.

Symbols used in the width operand must have been previously defined. (The phrase "must have been previously defined" will appear frequently throughout the remainder of this manual. Previously defined refers specifically to the EQU and SET statements and statement labels that assign a numerical value to a symbol. Hence, the phrase "must *not* have been previously defined" doesn't mean that the symbol may not have been used in a previous statement's operand such as a default operand. It means simply that the symbol must not have been previously assigned a numerical value with either the EQU or the SET statement or a statement label address assignment.)

Examples of INSTRUCTION statements:

```
INSTRUCTION WIDTH 48          "8080 EMULATOR" ;
INSTRUCTION WIDTH WORD_SIZE ;
```

The FIELD Statement

The FIELD statement assigns a mnemonic name to a specific Field and stipulates the number of bits in the Field. Optionally, a default value for the Field may be specified. The FIELD statement's format is illustrated in Figure 2.3.

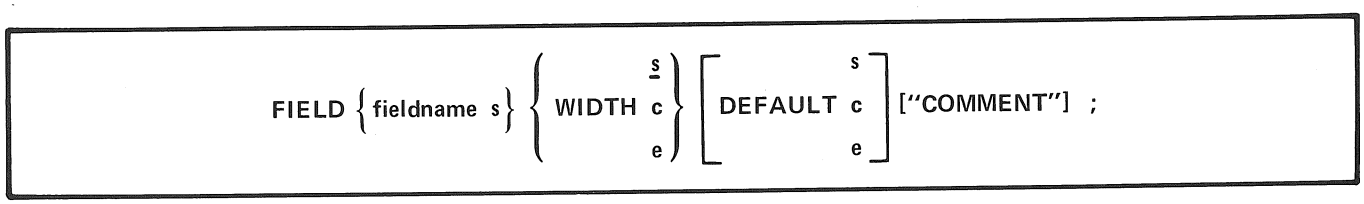


FIGURE 2.3

The FIELD statement begins with the FIELD Op-Code. Next, the Field is assigned a mnemonic name by the field name operand (the symbolic name must not have been used before).

Following the Field's name is the width operand. The width operand consists of the keyword WIDTH and a language element that the Micro Assembler can resolve to a numerical value. If a symbol is used in the width operand, it must have been previously defined in the user's source by an EQU or SET statement.

The optional default operand specifies a default value for the Field. If this option is exercised, the microprogram designer is freed from the responsibility of specifying the Field's value in every microinstruction. When the Micro Assembler assembles a microinstruction for which the Field value is not stipulated, the default value is placed in the microinstruction's object output. If the default option is not exercised, a value for the Field must be specified in each microinstruction of the microprogram.

The default operand consists of the keyword DEFAULT and a default operand. If the default operand contains symbols, the symbols need not have been previously defined (i.e., the symbol may be assigned a value later in the user's source).

Examples of FIELD statements:

```
FIELD OP WIDTH 7  DEFAULT NO_OP
  "FUNCTION FIELD FOR 3002" ;

FIELD DATA WIDTH 0FH  DEFAULT 0000H ;

FIELD BUFFER WIDTH BUFL ;

FIELD ADDR WIDTH 16  DEFAULT START+1
  "MICROPROGRAM ADDRESS" ;
```

Field Placement Within the Microinstruction

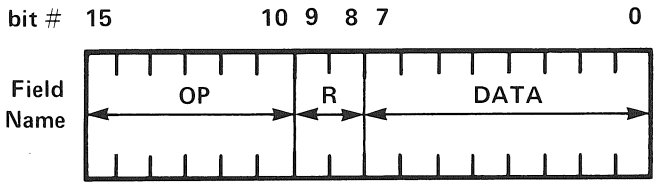
The Fields are placed in the microinstruction according to the order they are listed in the user's source. The first Field in the microinstruction is assigned to the high order bits of the microinstruction. Each successive Field is assigned to the highest order bits in the microinstruction not already used.

An example of Field placement for a 16-bit microinstruction is presented in Figure 2.4.

User Microinstruction Definition Source

```
INSTRUCTION WIDTH 16;
  FIELD OP WIDTH 6;
  FIELD R WIDTH 2;
  FIELD DATA WIDTH 8;
```

Diagram of Resulting Microinstruction



16 Bit Microinstruction

FIGURE 2.4

The total number of bits assigned to the various Fields must not exceed the microinstruction width. It is, however, permissible to leave bits allocated for the microinstruction unassigned to any Field. Microinstruction bits not assigned are unprogrammed. The Micro Format Program can set any unprogrammed bits in the microinstruction to default values.

END INSTRUCTION Statement

The END INSTRUCTION statement terminates the definition of a microinstruction. It consists of the reserved words: END INSTRUCTION. The END INSTRUCTION statement's format is given in Figure 2.5.

```
END INSTRUCTION        ["COMMENT"] ;
```

FIGURE 2.5

Microinstruction Definition Summary

The Microinstruction Definition portion of the microprogram designer's source contains three types of statements (INSTRUCTION, FIELD and END INSTRUCTION) which characterize the microinstruction and its composite Fields. An example of a complete Microinstruction Definition is presented in Figure 2.6.

```
INSTRUCTION WIDTH 16  "PRINTER
  CONTROLLER MICROINSTRUCTION" ;

    FIELD OP WIDTH 14  "FUNCTION CONTROL
      FIELD" ;

    FIELD R WIDTH 2  "DATA PORT SELECT
      FIELD" ;

END INSTRUCTION;
```

FIGURE 2.6

Microp Definition

Once the Microinstruction Definition portion of the source program has been written, it is possible to immediately begin the task of writing the microprogram itself. With the various Fields defined, a microinstruction can be symbolically represented as a sequence of Field names, each with an operand specifying its value for the microinstruction being written. This method is permissible and perfectly valid. However, the Micro Assembler provides another level of microprogramming convenience that makes the microprogram easier to write as well as easier to read. This extended level of microprogramming convenience is called the Microp.

A Microp is a mnemonic name for a specific value of a specific Field. Generally, the mnemonic name for a Microp relates to the Field's function when it contains the Microp's value. This feature not only lightens the burden of associating a Field's function with a numeric value; it also greatly enhances the readability of the microprogram source. The Microp is a powerful tool in the microprogram designer's arsenal of design techniques, and its utilization is strongly encouraged.

The MICROP Statement

A Microp is defined with the MICROP statement. The format of the MICROP statement is presented in Figure 2.7.

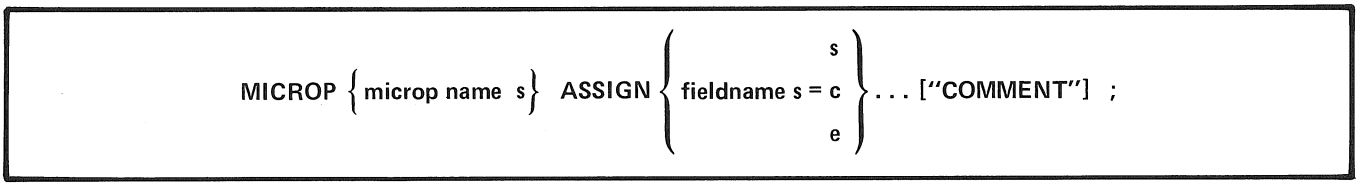


FIGURE 2.7

The MICROP statement begins with the MICROP Op-Code. This is followed by the Microp's symbolic name. The symbol used to name a Microp cannot have been previously defined. The Microp's name is followed by the keyword ASSIGN and the assign operand. The assign operand makes the actual Field value assignment. The Field referenced in the ASSIGN operand must have been defined in the Microinstruction Definition. If symbols are used to designate the Field's value, they need not have been previously defined. (Note the use of the "equal sign" (=) as a delimiter separating the Field name and Field value representation.)

When the Micro Assembler encounters a Microp in the microprogram source, it substitutes the entire ASSIGN operand for the Microp.

Examples of the MICROP Statements:

```
MICROP ADDI ASSIGN OP=21H "ADD IMMEDIATE
  FUNCTION

MICROP MEMW ASSIGN BUS=MEMORY    WRITE =
  TRUE

MICROP RESET ASSIGN ADDR=00H "JUMP TO
  START OF RST ROUTINE" ;
```

Intrinsic Microps

The primary purpose of the Signetics Micro Assembler is to support design efforts that include members of the Signetics bipolar Microprocessor Bit Slice family. Towards

that end, Microps that support the control fields for the N3002 Central Processing Element, the 8X02 Control Store Sequencer, and the N2901-1 Four-Bit Microprocessor Slice have been included as an intrinsic part of the Micro Assembler. These intrinsic Microps can be made part of the microprogram designer's Microinstruction Definition Section with the INTRINSIC Statement.

The INTRINSIC Statement

The INTRINSIC Statement requests that the Micro Assembler include the predefined Microps for a particular device in the user's Microinstruction Definition Section. The format of the INTRINSIC statement is given in Figure 2.8.

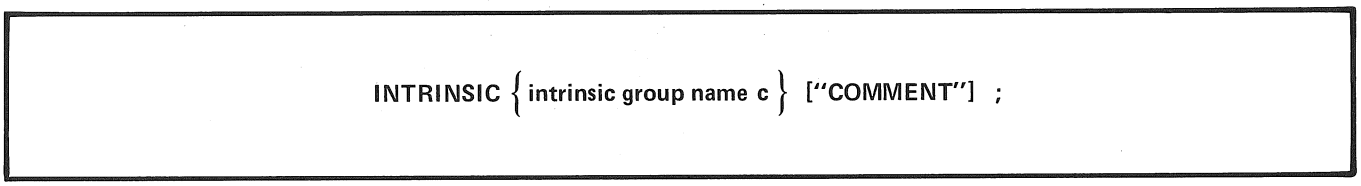


FIGURE 2.8

The INTRINSIC statement consists of the INTRINSIC Op-Code followed by a self-defining character constant that specifies the requested intrinsic group. As of this writing, three intrinsic groups are available:

- 3002
- 8X02
- 2901

As Signetics announces new bipolar Microprocessor devices, intrinsic groups of Microps supporting them will be made available to Micro Assembler users.

A full description of the intrinsic groups can be found in the Intrinsic Microp Section.

Examples of the INTRINSIC statement:

```
INTRINSIC       '3002' ;
INTRINSIC       '8X02' ;
INTRINSIC       '2901' ;
```

WRITING A MICROPROGRAM — THE MICROPROGRAM SECTION

Once the microprogram designer has defined the micro-inctruction, he is ready to write the microprogram itself. The microprogram is incorporated in the second major section of the source program, the Microprogram Section. The Microprogram Section first specifies the dimensions of the Control Store Memory where the microprogram will ultimately reside, and then lists the microprogram as a sequence of Microinstruction statements.

A Microinstruction statement specifies a value for each Field in the microinstruction. Utilizing the definitions provided in the Microinstruction Definition Section, the Micro Assembler assembles each Microinstruction statement and produces a line of intermediate object text. When the Micro Assembler has assembled the entire Microprogram Section, it generates the following outputs:

- The assembled microprogram as intermediate object text.
- A formatted listing of the source and object.
- A cross reference listing of all of the symbols used in the source.

The intermediate object text is used as source input to the Micro Format Program. The Micro Format Program processes the intermediate object text and outputs the final object output of the assembled microprogram. This object output is suitable for direct loading into the Control Store Memory.

As the Micro Assembler assembles the microprogram, it maintains the microprogram location counter. The location counter is initialized to zero by the first statement of the Microprogram Section. Thereafter it is incremented with each Microinstruction statement. The current value of the location counter is assigned to any address labels that occur within the microprogram source. The current value of the location counter also replaces the location counter reference symbol ($) when it is encountered by the Micro Assembler.

The current value of the location counter may be altered by the ORG statement. (The ORG statement will be described in the section dealing with directive statements.)

Directive statements may be freely interspersed with Microinstruction statements in the Microprogram Section. With the notable exception of the ORG statement, they have no effect on the location counter.

The PROGRAM Statement

The first statement of the Microprogram Section must be the PROGRAM statement. The PROGRAM statement specifies the length and width of the microprogram and hence the length and width of the Control Store Memory where it will reside. The PROGRAM statement also initializes the location counter to zero.

Figure 2.9 illustrates the PROGRAM statement's format.

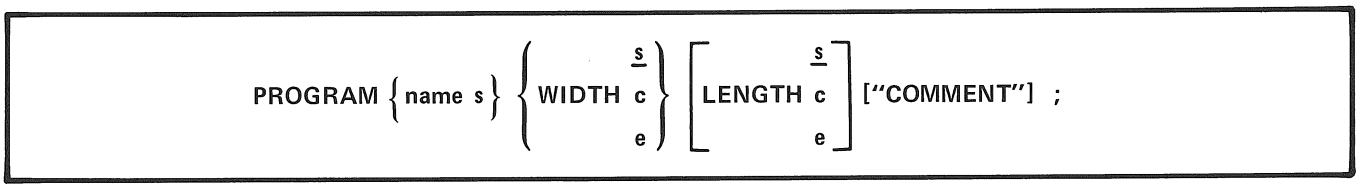


FIGURE 2.9

The PROGRAM statement consists of the PROGRAM Op-Code, a symbolic name, a width operand and an optional length operand.

The symbol used to define the PROGRAM statement must not have been previously used in the source program nor may it be used again (i.e., the symbolic name must be unique to the PROGRAM statement).

The width operand specifies the width of the Control Store Memory. The value it specifies is normally the same value assigned to the microinstruction with the INSTRUCTION statement. The width operand consists of the keyword WIDTH and a language element that is resolvable to a numerical value. If symbols are used to specify the width value, they must have been previously defined in the source program.

The length operand specifies the number of microinstructions that may be stored in the Control Store Memory (the length of the microprogram). The length operand consists of the keyword LENGTH and a language element representing the length value. If the length value is specified by a symbolic name, it must have been previously defined. When the length operand is omitted, the Micro Assembler assumes a length of 65,536 (16-bit address).

If the number of Microinstruction statements in the Microinstruction Section exceeds the value of the length operand, the Micro Assembler gives a memory overflow indication.

Examples of PROGRAM statements:

```
PROGRAM EMULATOR_8080 WIDTH 48 LENGTH
  512

PROGRAM DISC_CONTROLLER WIDTH 24
  LENGTH 128;

PROGRAM LONG_PROGRAM WIDTH 32;
```

Microinstruction Statements

The microprogram is written as a sequence of Microinstruction statements. For each microinstruction desired, there must be a corresponding Microinstruction statement. Clearly, the greatest amount of energy spent preparing the source program for the Micro Assembler will be spent writing Microinstruction statements.

The Microinstruction statement's format is presented in Figure 2.10.

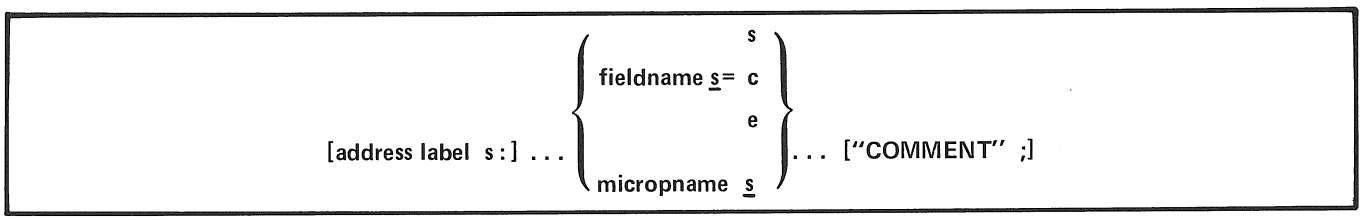


FIGURE 2.10

The Microinstruction statement has no Op-Code. The Micro Assembler recognizes a statement as a Microinstruction when it encounters a Field name or a Microp name.

If the microprogrammer wishes to do so, he may begin a Microinstruction statement with one or more address labels. Symbols used as address labels must not have been previously defined.

Following the address label (if one is used) is a list of Field assign operands and Microps that specify the value of each Field in the microinstruction. The Microinstruction statement must assign a value to every Field allocated in the Microinstruction Definition Section. Field values are determined by Field assign operands, Microps and default values specified by FIELD statements in the Microinstruction Definition Section. If a Field without a default value is not assigned, the Micro Assembler will give an error indication. An error indication is also given if the same Field within a Microinstruction statement is assigned more than one value.

Symbols used in Field assign operands need not have been previously defined.

Field assign operands and Microps may be listed in any order. However, it is strongly suggested that the microprogram designer select a format for Microinstruction statements and use it for each microinstruction. This approach greatly enhances the microprogram documentation value of the source listing.

Examples of Microinstruction Specification statements:

```
ADD_R1_ACC: FUNCTION=ADD  REG=R1
  KBUS=NOP  INCREMENT;

FETCH: F=NOP  MEM=REG  RCV=LATCH
  NEXT_OP_CODE;
```

The END Statement

The END statement terminates the Microprogram Section and hence the source input to the Micro Assembler. The END statement format is illustrated in Figure 2.11.

```
END          ["COMMENT"] ;
```

FIGURE 2.11

The END statement consists of the END Op-Code.

Example of the END statement:

```
END;
```

DIRECTIVES TO THE MICRO ASSEMBLER

The Micro Assembler directive statements are provided to both ease the task of writing a microprogram and improve the readability of the microprogram's documentation.

Assembly Directives

The EQU (Equate) Statement

The EQU statement equates a number to its symbolic name. EQU's format is illustrated in Figure 2.12.

The statement label is equated to the equate operand. The following statements are given as examples:

```
TRUE:ONE:HIGH:      EQU 1B;
FALSE:ZERO:LOW:     EQU 0B;
FIRST_ADDRESS:      EQU FIVE_BITS;
BUFFEND:            EQU BUFFER + BUFSIZE;
```

```
{label:} ...   EQU { s }  ["COMMENT"] ;
                   { c }
                   { e }
```

FIGURE 2.12

The EQU statement consists of one or more statement labels, the EQU Op-Code, and the equate operand. The statement label is the symbol to be given a numerical value. The label must not have been used before.

The equate operand is resolved to determine the label's numerical value. Any symbol used in an equate operand must have been previously defined. The equate operand may not reference the label it is defining (i.e., BUF: EQU BUF+1; is unacceptable).

The EQU statement makes a permanent value assignment. Any label defined by the EQU statement retains its assigned value for the entire assembly.

The SET Statement

In contrast to the EQU statement, the SET statement makes a value assignment that may be changed during the microprogram's assembly. Differing only in that aspect, the SET statement's function and format are identical to the EQU statement, as illustrated in Figure 2.13.

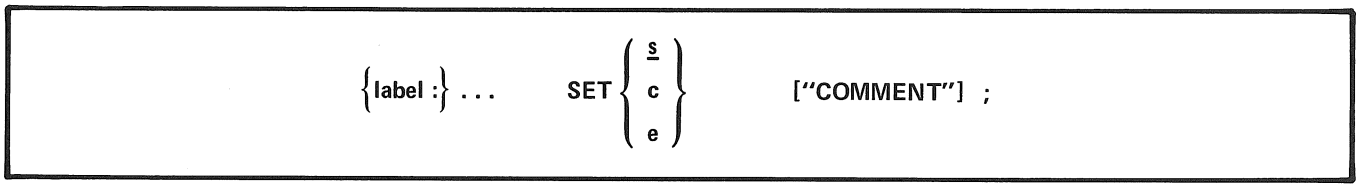


FIGURE 2.13

The SET statement consists of the symbolic label(s), the SET Op-Code and the set operand.

Unlike the EQU statement, the label of the SET statement may have been defined earlier in the source text. Previous definition must be made by another SET statement however.

Symbols used in the SET operand must have been previously defined and be resolvable to a numerical value.

Figure 2.14 illustrates an example of SET's utility. Control Field A and Control Field B have functions that are mutually exclusive in time. This situation permits saving microinstruction bits without affecting the horizontal nature of the microprogram, by demultiplexing a Field within the microinstruction (Figure 2.14 B). The one bit control field SEL determines which control field is active.

In microcode, this scheme is implemented by dividing the microprogram into two sections: one section servicing Control Field A and another section servicing Control Field B. It is quite likely that the two control fields have different default codes. The Micro Assembler will support both default conditions if the Field called CONTROL is given a symbolic default (e.g., NOP) and if NOP is assigned different values in each section of the microprogram with the SET statement (Figure 2.14 A).

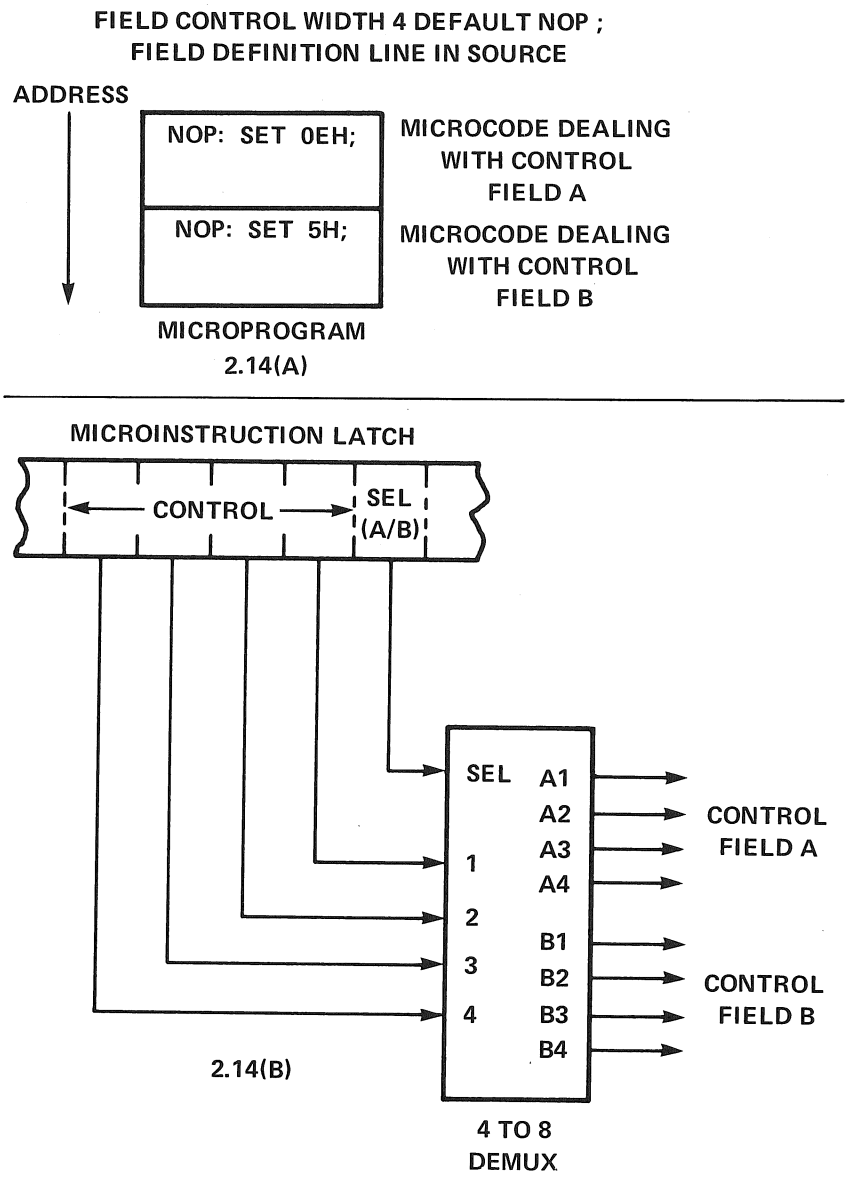


FIGURE 2.14

The ORG Statement

The ORG statement allows the microprogrammer to organize his microprogram source in a non-sequential format. The ORG statement provides this flexibility by assigning a new value to the current program location counter. The ORG statement's format is given in Figure 2.15.

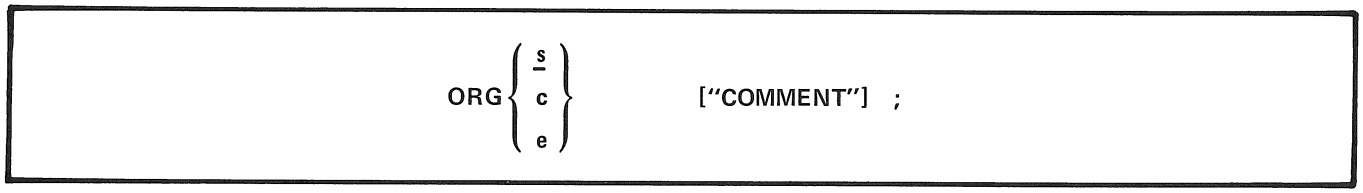


FIGURE 2.15

The ORG statement consists of the ORG Op-Code followed by the ORG operand. The ORG operand represents the new location counter value that is to be substituted for the current one. Symbols used in the ORG operand must have been previously defined.

It is permissible to reference the current value of the location counter within the ORG operand. This is done in the normal way, by using the ($) symbol.

Examples of the ORG Statement:

```
ORG  40H     "BEGIN MULTIPLY SEQUENCE" ;
ORG  START   "PROGRAM ORIGIN" ;
ORG  $+2     "SKIP TWO WORDS" ;
```

Listing Directives

The second type of assembler directive allows the microprogrammer to control the format of the Micro Assembler's output listings.

The LIST Statement

The LIST statement controls the listing output produced by the Micro Assembler after the microprogram has been assembled. Figure 2.16 presents the LIST statement format.

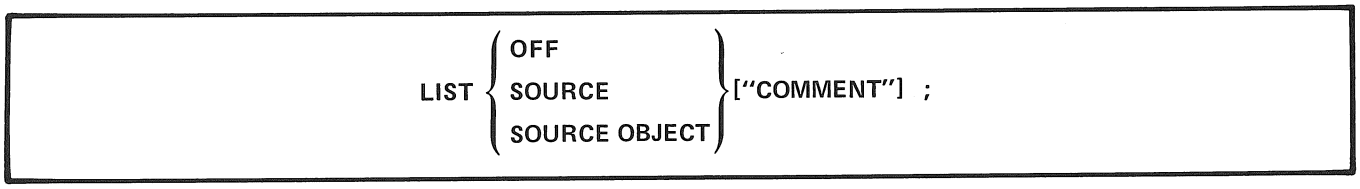


FIGURE 2.16

The LIST statement begins with the LIST Op-Code and ends with one or two reserved words as the LIST operand. The reserved words instruct the Micro Assembler as to which listings should be generated during the assembly process. The reserved words and their respective functions are listed below.

List Operand	Function
OFF	Suppresses the source/object listing from line where LIST statement appears.
SOURCE	Permits the generation of the source listing without accompanying object code from line where LIST statement appears.
SOURCE OBJECT	Allows the source/object listing to be produced beginning at line where LIST statement appears.

If the LIST statement is not included in the user's source, the Micro Assembler assumes a default of SOURCE OBJECT.

The OBJECT Statement

The OBJECT statement controls the generation of the intermediate object text. The format of the OBJECT statement is given in Figure 2.17.

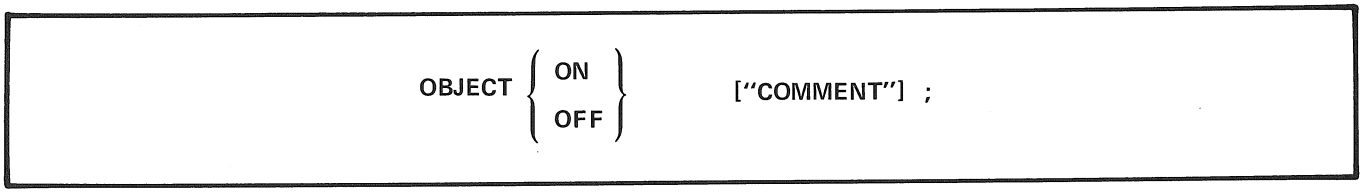


FIGURE 2.17

The OBJECT statement consists of the OBJECT Op-Code followed by a reserved word. The reserved word may be either ON or OFF. Their respective meanings are as follows:

ON — Indicates that the intermediate object text should be produced.

OFF — Indicates that the intermediate object text is to be suppressed.

As within the LIST statement, the OBJECT statement may be interspersed throughout the source program. If the OBJECT statement is not included in the source, it defaults to ON.

The SPACE Statement

The SPACE statement produces blank lines in the source and object listing. The SPACE statement has the following format.

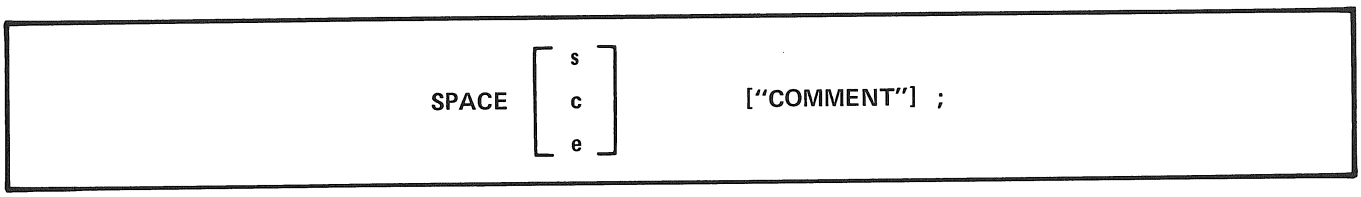


FIGURE 2.18

The SPACE statement consists of the SPACE Op-Code and the space operand which the Micro Assembler resolves to a numerical value. This numerical value is the number of blank lines inserted in the source/object listing. Symbols used in the space operand need not have been previously defined. If the space operand is omitted, only one blank line is produced. The SPACE statement is not reproduced in the source/object listing.

THE EJECT Statement

The EJECT statement terminates the listing output on a given page and ejects the remainder of the page. Its format is illustrated in Figure 2.19.

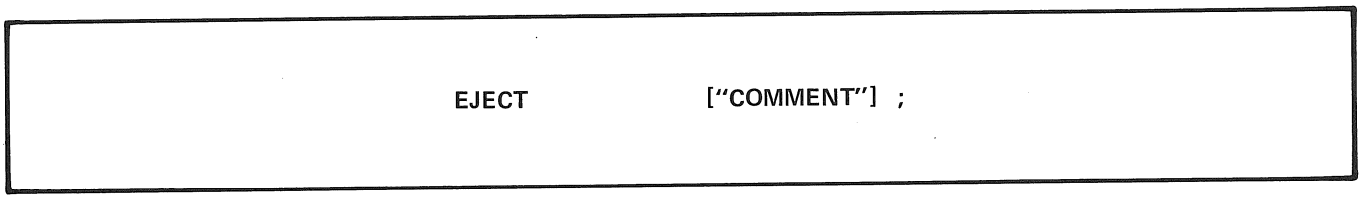


FIGURE 2.19

The EJECT statement consists only of the EJECT Op-Code itself.

When the Micro Assembler encounters the EJECT statement, it ejects the remainder of the current listing page and begins the next page with the normal source/object listing page heading. The EJECT statement is not reproduced in the source/object listing.

The TITLE Statement

The TITLE statement places text in the user defined portion of the source/object listing page heading. Figure 2.20 presents the TITLE statement's format.

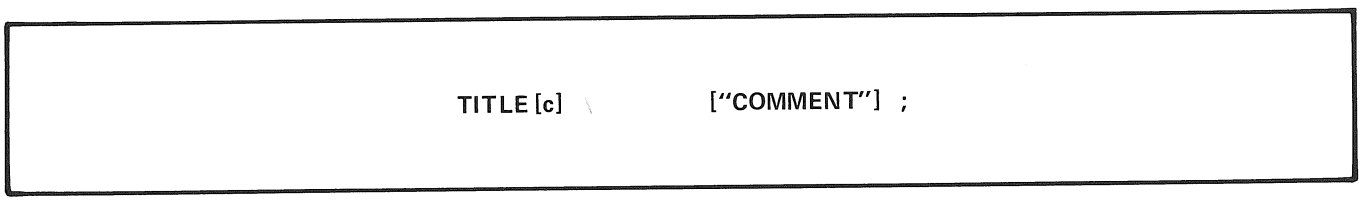


FIGURE 2.20

The TITLE statement consists of the TITLE Op-Code and an optional character constant. The user text portion of the source/object listing heading is limited to 28 characters (in this instance, the character constant size is not limited by the M toggle). If the optional character constant is omitted, the user text is filled with blanks. When the Micro Assembler encounters a TITLE statement, it defines (or redefines) the user text in the heading, and ejects the remainder of the current listing page. The TITLE statement is not reproduced on the source/object listing.

PART THREE – EXTENDED FEATURES

The power of the Micro Assembler as a microprogramming tool is greatly enhanced by the extended features. The extended features include:

- Additional logical and relational expression operators
- Facilities for programming non-microinstruction PROMs (e.g., Field expansion, mask generation and look-up table PROMs)
- Microp defaults
- Microp arguments
- An IF, THEN, ELSE clause for Microps
- Sub-Field definition
- Multiple Field definition
- Multiple Microinstruction format definition

EXPRESSION EXTENSIONS

The set of operators that may be used in expressions is expanded to include logical, shift and arithmetic comparison operations.

Logical Operators

The logical operators perform Boolean functions on the operands within an expression. With the exception of the logical not function, all of the logical operators require two operands. The name and function of the logical operators is given below:

TABLE 3.1

Name (reserved word)	Function
OR	Inclusive OR of two operands.
XOR	Exclusive OR of two operands.
AND	Logical AND of two operands.
NOT	Complements the operand it precedes.*

*Note: (If the number of bits in the operand is less than the maximum word size of the Micro Assembler (set by M toggle), high order zeros are appended to the operand until it is the maximum word size. This expanded value is then complemented. High order bits are then truncated when the complemented value is assigned to a Field of fewer bits than the maximum word size.)

Shift Operators

The shift operators shift an operand by a number of bits specified by the user. The shift operator names and functions are listed in Table 3.2 below.

TABLE 3.2

Name (reserved word)	Function
SHL	Shift Left shifts the first operand left by the number of bits specified with the second operand. Zeros are shifted into the low order bits. Bits shifted out of the most significant bit are discarded. (The most significant bit is determined by the maximum word size set by the M toggle.)
SHR	Shift Right shifts the first operand right by the number of bits specified with the second operand. Low order bits shifted out are discarded. Zeros are shifted into the high order bits.

Shift Examples:

1000_1001B SHL 2 results in ...10_0010_0100B

1000_1001B SHR 3 results in ...00_0001_0001B

Arithmetic Comparison Operators

The arithmetic comparison operators compare the arithmetic values of two operands (negative numbers are in two's complement form). If the comparison is true, the result is all ones (arithmetic negative one). Conversely, if the comparison is false, the result is all zeros. The arithmetic comparison operators are used in conjunction with IF THEN ELSE statements to be discussed later. The following table names the arithmetic comparison operators and describes their operation.

TABLE 3.3

Name (reserved word)	Function
EQ	Equal — The results of this operation are true if the two operands are equal.
NE	Not Equal — The results of this operation are true if the two operands are not equal.
GT	Greater Than — The results of this comparison are true if the first operand is greater than the second operand.
GE	Greater Than or Equal — The results of this operation are true if the first operand is greater than or equal to the second operand.
LT	Less Than — The results of this comparison are true if the first operand is less than the second operand.
LE	Less Than or Equal — The results of this operation are true if the first operand is less than or equal to the second operand.

Operator Evaluation Hierarchy

When added to the basic operators, the extended operators have the following evaluation hierarchy:

1. SHL, SHR
2. +, -
3. EQ, NE, GT, GE, LT, LE
4. NOT
5. AND
6. OR, XOR

Operators are evaluated from the top of the list (i.e., the shift operators are evaluated first, the add and subtract operators second, etc.). Operators on the same line are evaluated left to right as they appear in an expression. As with the basic assembler, this hierarchy can be modified with parentheses.

Examples of expressions with the extended operators:

Expression	Order of Evaluation
BUF GT B + 4	BUF GT (B+4)
REG GE 0 AND REG LE 11	(REG GE 0) AND (REG LE 11)
1H OR INPUT AND 1F0H	1H OR (INPUT AND 1F0H)
(BUF+1) SHL 3 + BASE	((BUF+1) SHL 3) + BASE

MULTIPLE MEMORY BLOCKS

The Micro Assembler supports assembly of data fields for control PROMs that the microprogram designer may wish to add to his basic microprogram based system. Additional control PROM Program Sections (such as mask tables, jump address tables and character tables), share symbol references with the main microprogram. For example, a symbolic address for a control PROM may be assigned as a field value in the microprogram.

Additional Program Sections are included in the user's source with the same format as the original microprogram Program Section. An additional memory block begins with a PROGRAM statement, and is terminated with the next PROGRAM statement. The last memory block is terminated with an END statement.

Depending on the complexity of the Program Section, it may require a Microinstruction Definition Section, or simply be assembled with DCL statements.

The DCL (Declare) Statement

The DCL statement specifies data for an auxiliary memory block. The format of the DCL statement is presented in Figure 3.1.

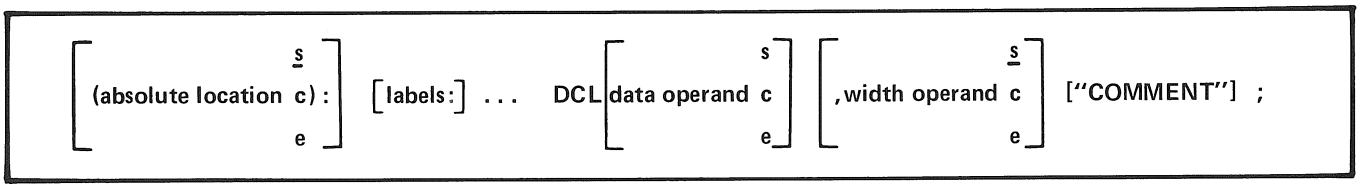


FIGURE 3.1

Figure 3.1 introduces a new language element, the absolute location label. The absolute location label overrides the microprogram location counter with a specified numerical value. A full discussion of the rules governing the use of the absolute location label is presented in the next section.

The DCL statement begins with optional statement labels. If the statement does not have an absolute location label, it is given the current value of the location counter as an address. Following the statement label(s) is the DCL Op-Code. The DCL Op-Code precedes the data value operand. The data value operand is resolved by the Micro Assembler into object data that will be placed in the control PROM. Symbols used in the data value operand need not be previously defined. Character constants used as data values may exceed the M toggle. Following the data operand is the optional width operand. Symbols used in the width operand must have been previously defined. If the width operand is included, it must be separated from the data operand by a comma. If the width operand is omitted, the data width is defaulted to WIDTH value specified in the PROGRAM statement. If the data width specified is wider than the PROGRAM WIDTH, it is divided into multiple words. Zeros are inserted into the high order bits of the most significant word to form a complete data word. If, however, the data value is wider than the width operand, the high order bits are truncated with no error indication.

The data value operand is optional. If the data value is omitted, no data is placed in the memory location. In this case, the DCL statement acts as a space reservation.

Examples of the DCL Statement:

```
BEGIN: DCL 'ERROR', 40   "ERROR MESSAGE
                          HEADER" ;

(0FH): DCL 5H            "REGISTER MASK" ;

       DCL       , 16    "RESERVE TWO
                          BYTES" ;
```

Absolute Location Labels

Absolute location labels allow the microprogrammer to override the location counter assignment as a microinstruction or DCL statement address. Statements with absolute location labels do not affect (increment) the program counter. The Micro assembler places the object data at the location specified.

As the location counter is not affected by absolute location address assignments, care must be taken not to assign two microinstructions to the same Control Store address.

When the absolute location label is used, it must be the first language element in the statement. It is preceded by a left parenthesis and is terminated by a right parenthesis followed by a colon. One or more symbolic name labels may follow the absolute location label and they are assigned the value of the absolute location label.

The absolute location label may be a symbol, a self-defining constant or an expression. If symbols are used in the label, they must have been previously defined.

Absolute location labels may only be used with a Microinstruction statement or a DCL statement.

Examples of Absolute Location Labels:

```
(80H):
(512):
(MOVE):
(MOVE + 40H):
```

MICROP EXTENSIONS

In Part Two, a Microp was defined as a symbolic name for a specific Field value. In this section we will expand that definition to include multiple Field assignments, Microp default assignments, conditional Field value assignments, Field value assignment validation and Microp arguments. Clearly, the Microp extensions provide the microprogrammer with powerful microprogramming facilities.

Microp Defaults

Remember that a microinstruction control Field may be assigned a default value in the defining FIELD statement. This default value will be assigned to the Field in those Microinstruction statements that make no specific reference to the Field.

The Microp DEFAULT operand allows the microprogrammer to temporarily change a Field's default value as the microprogram is being assembled. When the DEFAULT operand is specified, it is appended to a normal MICROP statement. The DEFAULT operand consists of the keyword DEFAULT and one or more Field assign operands. Microp DEFAULT Field assign operands do not assign values to Fields as does the ASSIGN operand. When a Microp with DEFAULT operands is included in a Microinstruction specification statement, the default Field assign operands temporarily override the normal default value of the referenced Field.

The default operates when a microinstruction contains its defining Microp. If the microinstruction contains no other reference to a Microp defaulted Field, it is assigned the temporary default value.

The following Figure illustrates the use of Microp defaults.

Statements in the Microinstruction Definition Section

```
INSTRUCTION WIDTH 5 ;

   FIELD X WIDTH 2 ;

   FIELD Y WIDTH 3 DEFAULT 100B  "NORMAL
     DEFAULT FOR Y IS 4" ;

END INSTRUCTION ;

MICROP SETX ASSIGN X=11B  DEFAULT  Y=001B
   "TEMPORARY Y DEFAULT=1";

MICROP SETY ASSIGN Y=010B ;
```

Statements in the Program Section

```
PROGRAM CONTROL_TABLE WIDTH 5 ;

   X=0              "X=00B, Y=100B" ;
   SETX             "X=11B, Y=001B" ;
   SETX Y=111B      "X=11B, Y=111B" ;
   SETX SETY        "X=11B, Y=010B" ;
END;
```

FIGURE 3.2

If two Microps which have DEFAULT Field assignments for the same Field are used in the same microinstruction, and the field is otherwise unreferenced in that microinstruction, an error indication will be given.

Microp Arguments

The MICROP statement is extended to allow definition of Microp arguments. Microp arguments allow the microprogrammer to pass values to the Microp when it is used in a Microinstruction statement. The referenced Microp uses the values to compute Field values and make the proper Field assignments.

When a Microp includes arguments, the Microp name is followed by one or more arguments separated by commas and enclosed in parentheses. The arguments may be symbols or expressions. Once arguments have been associated with a Microp, any Microinstruction statement may reference the Microp with a list of values that are substituted for the arguments. The form of the Microp used in a Microinstruction statement is similar to the format of the MICROP statement that originally specified the arguments. The Microp name is followed by a left parenthesis, and a list of values to be substituted for the arguments. The argument value assignment is made by positional correspondence (i.e., the first value is assigned to the first argument named in the defining MICROP statement, the second value to the second argument named and so on). The argument values are separated by commas and are terminated by a right parenthesis.

The following is given as an example:

Defining MICROP Statement

```
MICROP MOVE (REG_SOURCE, REG_DEST)
   ASSIGN OP=20H SOURCE=REG_SOURCE
   DEST=REG_DEST ;
```

Microinstruction Utilizing MOVE Microp

```
MOVE (2,11)    "ASSIGNS OP=20H, SOURCE=2
                  & DEST=11" ;
```

Symbols used as argument names may not be used as value symbols or statement labels elsewhere in the user's source. However, argument symbols are local to their associated Microp and can therefore be used in more than one Microp.

Arguments may have default values. Default values are specified in the MICROP statement that defines the argument. An argument is assigned a default by following the argument name with an equal sign (=) and the desired default values. If the Microp is used in a Microinstruction statement without specifying a value for the argument, the default value is substituted for the argument name in the Microp.

Example of MICROP Statement with Argument Default

```
MICROP OP (OP_SELECT=08H)        ASSIGN
   FUNCTION=OP_SELECT  "OP DEFAULT=NOP" ;
```

If the following microinstruction is specified,

```
OP  R(ACC,ACC)  MEMOFF   "THIS INSTRUCTION
   IS NOP" ;
```

the OP Microp assigns 08H to the microinstruction Field FUNCTION.

The argument structure of Microps allows the Micro Assembler to be used as a meta assembler. Toward that end, additional argument syntaxes are supported. There are two basic argument formats. The form discussed so far is a subset of one of these formats.

Argument Format One

The first argument format looks very much like a Field assign operand. It consists of the Microp name followed by an equal sign (=) and a single argument. Within a defining MICROP statement, argument Format One would be expressed as follows:

```
MICROP    TERM=FACTOR    ASSIGN FIELD_1=
  (FACTOR SHL 1) ;
```

Microinstruction statements including the Microp TERM would have the corresponding formats illustrated below:

```
ADD(R2, R5)   TERM=0EH   ADDRESS(NEXT) ;

ADD(R2, R5)   TERM=(0FF H AND MASK)
  ADDRESS(NEXT) ;
```

In the above examples, the Microp ADD and ADDRESS has the argument format described in the beginning of this section and the Microp TERM has argument Format One.

Argument Format One does not allow Microp default operands.

Argument Format Two

Argument Format Two allows argument lists in several forms. These forms are listed below.

1. microp-name (operand-list)
2. microp-name operand list
3. microp-name (operand-list) operand-list

The operand list in the above forms is one or more argument operands separated by commas. The argument format first presented in this section was number (1) above.

An argument operand has the following formats:

a. argument

b. argument (operand-list)

c. (operand-list)

The basic element of an argument operand is an argument. In the defining MICROP statement this is the argument name and in a referencing microinstruction it is the value being passed to the Microp. Note that form (c) above implies recursive argument definition. That is, an argument operand in an argument list may itself be an argument list. There is no theoretical limit to argument recursion.

When Microps defined with argument Format Two are referenced, argument operands may be omitted. If the omitted arguments are within a list, their place must be kept by commas. If, however, all of the omitted arguments are at the end of an argument list, the place-keeping commas may also be omitted.

When an argument list is enclosed by parentheses and either the entire list or all but the first argument are omitted, the parentheses may also be omitted. The only exception to this rule is with form (3) above. When form (3) is used, the parentheses enclosing the first operand-list must be included even if the operand-list is omitted.

Examples of Argument Format Two

Defining MICROP Statement	Format
MICROP ADD(R1, R2)	1 - c
MICROP ADD R1,ADDR(LENG,INDEX)	2 - b
MICROP ADD(R1, R2) CARRY(IN,OUT),OVF	3 - b & c

Arguments defined for a Microp may be used in that Microp as elements of value expressions in ASSIGN operands and Microp default operands. Argument Format Two allows the arguments themselves to have defaults as described in the beginning of the Microp argument discussion. Figure 3.3 presents some examples of Microp arguments.

```
                "2650 MICROPS"
                MICROP STRZ REG ASSIGN OP1=60Q RV1=REG;
                MICROP SUBI(REG) LITERAL ASSIGN OP2=51Q RV2=REG OPND2=LITERAL;
                I:  EQU  1B  "INDIRECT ADDRESSES";
                MICROP BSTA(CC) ADDR(INDIRECT=0)
                   ASSIGN OP3=17Q RV3=CC INDIR3=INDIRECT ADD3=ADDR;
                INCR:  EQU 01B  "INDEX CONTROL:  AUTO-INCREMENT";
                DECR:  EQU 10B  "INDEX CONTROL:  AUTO-DECREMENT";
                MICROP IORA(REG) ADDR(INDIRECT=0),INDEX=-1,CTL=0
                   ASSIGN IF INDEX EQ -1 THEN RV3=REG XCTL=0
                      ELSE IF CTL GT 0 THEN XCTL=CTL
                         ELSE XCTL=11B FI
                      RV3=INDEX FI
                   OP3=33Q INDIR3=INDIRECT XADD=ADDR;
                PROGRAM MEM WIDTH 8 LENGTH 256  "2650 PROM";
                ORG          0;
R0:             EQU          0;
R1:             EQU          1;
R2:             EQU          2;
R3:             EQU          3;
CEQ:            EQU          0;
CLT:            EQU          1;
CGT:            EQU          2;
CUN:            EQU          3;
START:          STRZ         R2;
                SUBI(R2)     1;
                BSTA(CEQ)    RETADDR(I);
                IORA(R0)     DATA,R3,INCR;
```

FIGURE 3.3

Microps within Microps

Microps may be nested. That is, a Microp may reference other Microps. This feature allows a Microp to be used in a variety of ways. Not only may a Microp be used as a mnemonic name for a specific Field value, but it may define values for a number of Fields. In fact, a Microp may specify Field values for an entire microinstruction.

If a Microp is being defined with other Microps, the referenced Microps must have been defined earlier in the source program with MICROP statements. When a Microp is referenced in the ASSIGN operand of another Microp, it is equivalent to including the Field assignments of the referenced Microp in the ASSIGN operand of the referencing Microp. Likewise, the Default operand of the referenced Microp is included in the DEFAULT operand of the referencing Microp.

The following example illustrates a nested Microp.

Statements from the Microinstruction Section

```
MICROP MEMON(DIRECTION)
   ASSIGN MEM=DIRECTION ;

MICROP DBUSON(DIRECTION)
   ASSIGN DB=DIRECTION   ;

      IN: EQU 01B ;

      OUT: EQU 10B ;

MICROP FETCH ASSIGN MEMON(OUT)
   DBUSON(IN)              ;
```

In the above example, the Microp FETCH assigns a value of 10B to the control Field MEM and a value of 01B to the control Field DB. (Note: The Micro Assembler suffers no confusion over the multiple use of the symbol DIRECTION. Microp argument symbols are defined locally to the Microp that references them.)

When a Microp is referenced in the DEFAULT operand of another Microp, both the ASSIGN and DEFAULT operands of the referenced Microp are included in the DEFAULT operand of the referencing Microp. In the following example, FETCH5 and FETCH6 produce identical results.

Microinstruction Definition Section Statements

```
MICROP FETCH5 ASSIGN A=FETCH DEFAULT
   BRT NOP ;

MICROP BRANCH ASSIGN BRT DEFAULT
   NOP ;

MICROP FETCH 6 ASSIGN A=FETCH DEFAULT
   BRANCH ;
```

The IF Clause

As defined so far, the ASSIGN and DEFAULT operands of MICROP statements are comprised of a list of Field assign operands and Microp references. This sub-section will introduce one last language element that may be included in Microp ASSIGN and DEFAULT operands: the IF clause.

The IF clause provides conditional Field assignment and Microp referencing. This facility may be used to actually compose microinstructions or to test the validity of Field assignments.

The IF clause begins with the keyword IF, and is terminated by the keyword FI. Following the IF keyword is the IF operand, the THEN operand, an optional ELSE operand and finally, the terminating keyword FI.

The IF Operand

The IF operand is resolved to a number. The number is then compared to zero. If the value is non-zero, the test is considered true and the IF clause proceeds with the THEN operand. If, on the other hand, the Micro Assembler resolves the IF operand to zero, the test is considered false and the IF clause continues with the ELSE operand. This action is illustrated in Figure 3.4.

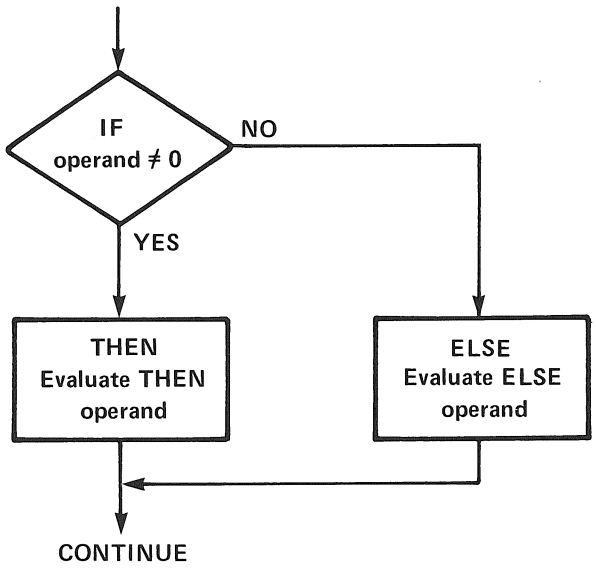


FIGURE 3.4

The IF operand value is usually the result of relational operations or of logical operations on relational results.

Typical IF Operands

IF REG GE 10	"IF REGISTER IS GREATER THAN OR EQUAL TO 10" (If REG≥ 10, the result is non-zero and the test is true.)
IF ADDRESS LT 128	"IF ADDRESS IS LESS THAN 128"

The THEN Operand

The THEN Operand must follow the IF Operand. The THEN Operand consists of the keyword THEN and a list of Field assignments and possibly nested Microps. A further level of testing may occur in a THEN operand by including another IF clause (either directly or with another Microp).

Typical THEN Operands

THEN ADD(R3, R7)	"MICROINSTRUCTION ADDS R3 TO R7"
THEN NOP	"MICROINSTRUCTION IS NOP"
THEN ADDRESS_TEST	"MICRO ASSEMBLER EVALUATES ADDRESS_TEST"

The ELSE Operand

The optional ELSE operand consists of the keyword ELSE, and Field assignments, Microps or further IF clauses. If the ELSE Operand is omitted and the IF Operand comparison is false, an error message will be generated indicating an invalid reference (e.g., assigning five bits to a 2-bit field).

Typical IF Clauses

```
IF REG GE 10  THEN F = 22H+REG
   ELSE F=20H + REG  FI

IF(R EQ AC)OR(R EQ T)  THEN F = 74H + R
  "ERROR IF NOT AC,T"  FI
```

MULTIPLE MICROINSTRUCTION FORMATS

The Micro Assembler as described so far has emphasized a horizontal approach to microprogramming. Each statement described has had as its ultimate goal the generation of specific bit patterns for specific fields. This subsection will present features that support a vertical approach by allowing Micro Control Store bits to service more than one control Field. The trade-offs between horizontal and vertical microprogramming techniques have been discussed at length in the literature and will not be considered here. Our aim is simply to provide the microprogrammer with the alternative of selecting either technique.

Vertical microprogramming is supported with multiple microinstruction formats. Multiple microinstruction formats may be accomplished in one of two ways. The first way is by defining sub-Fields. The other way is to define more than one microinstruction by writing multiple Microinstruction Definition Sections in the source program. Both of these methods will be discussed.

Sub-Field Definition

Any Field within a microinstruction may be divided into sub-Fields. Sub-Fields are defined with FORMAT statements. The procedure is to define the major Field with a FIELD statement and then define the sub-Fields with a FORMAT statement followed by FIELD statements that define the sub-Fields, and finally terminate the major Field definition with the END FORMAT statement. This procedure is illustrated in Figure 3.5.

Microinstruction Definition Section Statements

```
FIELD N3002_ARRAY_FUNCTION WIDTH 15          "CPE ARRAY HAS FOUR 3002S";
    FORMAT;
        FIELD KB WIDTH 8          "FOUR 3002S REQUIRE 8 K-BUS INPUTS";
        FIELD F  WIDTH 7          "CPE ARRAY SHARES COMMON F-BUS";
          FORMAT "SUB FIELDS OF F";
              FIELD F_GROUP WIDTH 3;
              FIELD R_GROUP WIDTH 4;
          END FORMAT;
    END FORMAT;
```

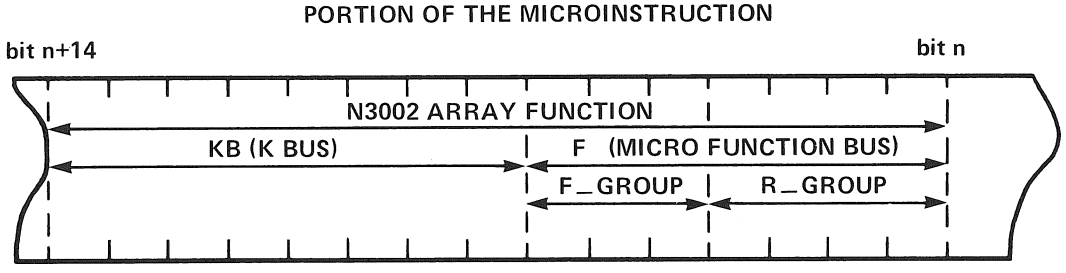


FIGURE 3.5

As can be seen in Figure 3.5, sub-Fields are assigned within a Field from left to right. The first sub-Field defined going in the highest-order bits of the major Field, and so on.

Returning to the concept of vertical microprogramming, a Field may have more than one set of sub-Fields. This is accomplished by including more than one sub-Field definition section. Each sub-Field definition section is bounded by a FORMAT statement and an END FORMAT statement. An example of multiple sub-Field definitions is presented in Figure 3.6.

Portion of Microinstruction Definition Section

```
FIELD FUNCTION WIDTH 11      "MAJOR FIELD DEFINITION";

        FORMAT  "THIS IS FOR N3002 FUNCTION CONTROL" ;
                FIELD F   WIDTH 7      "N3002 MICRO FUNCTION CONTROL FIELD";
                FIELD KB WIDTH 3       "K BUS PROM ADDRESS";
                FIELD CLK WIDTH 1  DEFAULT 1     "ENABLE N3002 CLK";
        END FORMAT;

        FORMAT  "THIS IS FOR 8X02 BRANCHES";
                FIELD A  WIDTH 10        "8X02 ADDRESS INPUT";
                FIELD NO_CLK WIDTH 1  DEFAULT 0  "INHIBIT N3002 CLK";
        END FORMAT;
```

FIGURE 3.6

In the case of a multiple format Field, the Micro Assembler selects the Field's format on the basis of the sub-Fields referenced in the Microinstruction statement. If sub-Fields from more than one format are used in a single microinstruction, an error indication is given.

As with normal Fields, sub-Fields without default values must be assigned a value every time their format is used.

In a Microinstruction statement, an instruction field with sub-Fields may also be assigned a value using the Field name. In this case, the sub-Field names may not be referenced. When neither the Field name nor the sub-Field names are referenced in a Microinstruction statement, the Field default is used.

Multiple Microinstruction Formats

More than one microinstruction may be defined in the Microinstruction Definition Section of the source. Each microinstruction defined requires an INSTRUCTION statement, associated FIELD statements and a terminating END INSTRUCTION statement. MICROP and directive statements may be included within instruction definitions.

This feature allows more than one format for a microinstruction. It also allows the width of the microinstruction to vary according to the format chosen. A given Microinstruction statement may use any one of the formats defined. The format chosen is determined by the Field names referenced. If Field names from different formats are used in the same microinstruction, an error indication is given.

An example of multiple microinstruction definition is given in Figure 3.7 (MICROPS defined in Figure 3.3).

```
"2650 INSTRUCTIONS"
INSTRUCTION WIDTH 8; "ONE BYTE INSTRUCTIONS"
        FIELD OP1 WIDTH 6;
        FIELD RV1 WIDTH 2;
END INSTRUCTION;
INSTRUCTION WIDTH 16;  "TWO BYTE INSTRUCTIONS"
        FIELD OP2 WIDTH 6;
        FIELD RV2 WIDTH 2;
        FIELD OPND2 WIDTH 8;
                FORMAT;
                        FIELD INDIR2 WIDTH 1;
                        FIELD ADD2 WIDTH 7;
                END FORMAT;
END INSTRUCTION;
INSTRUCTION WIDTH 24;  "THREE BYTE INSTRUCTIONS"
        FIELD OP3 WIDTH 6;
        FIELD RV3 WIDTH 2;
        FIELD OPND3 WIDTH 16;
                FORMAT;
                        FIELD INDIR3 WIDTH 1;
                        FIELD ADD3 WIDTH 15;
                          FORMAT;
                                FIELD XCTL WIDTH 2;
                                FIELD XADD WIDTH 13;
                          END FORMAT;
                END FORMAT;
END INSTRUCTION;
```

FIGURE 3.7

INTRINSIC MICROPS

The Micro Assembler supports Signetics' bipolar LSI microprocessor elements by providing predefined intrinsic microps for the LSI elements' control fields. As of this writing, three devices are supported with intrinsic microps. These are the 8X02 microprogram sequencer, the N2901-1 4-bit RALU (Register Arithmetic Logic Unit) slice and the N3002 2-bit RALU slice. The intrinsic microps for each device are included in the microprogrammer's Microinstruction Definition Section with an INTRINSIC Statement.

The intrinsic microps provide both value symbols (defined with EQU Statements) and predefined microps for use in writing the Microprogram Section. The microps make value assignments to undefined fields. That is, they make value assignments to field names but the intrinsic definitions do

not include FIELD Statements defining those fields. This has been left to the microprogrammer so that the fields may be assigned anywhere within the microinstruction. However, the fields must be defined prior to the INTRINSIC Statement that references them and they must be defined with the same mnemonic names defined in the following intrinsic microp descriptions. To avoid conflict with user mnemonics, all symbols internal to the intrinsic microps begin with the at sign (@).

8X02 Microps

The Signetics 8X02 is a microprogram sequencer capable of addressing 1,024 microinstructions. A block diagram of the device is presented in Figure 3.8.

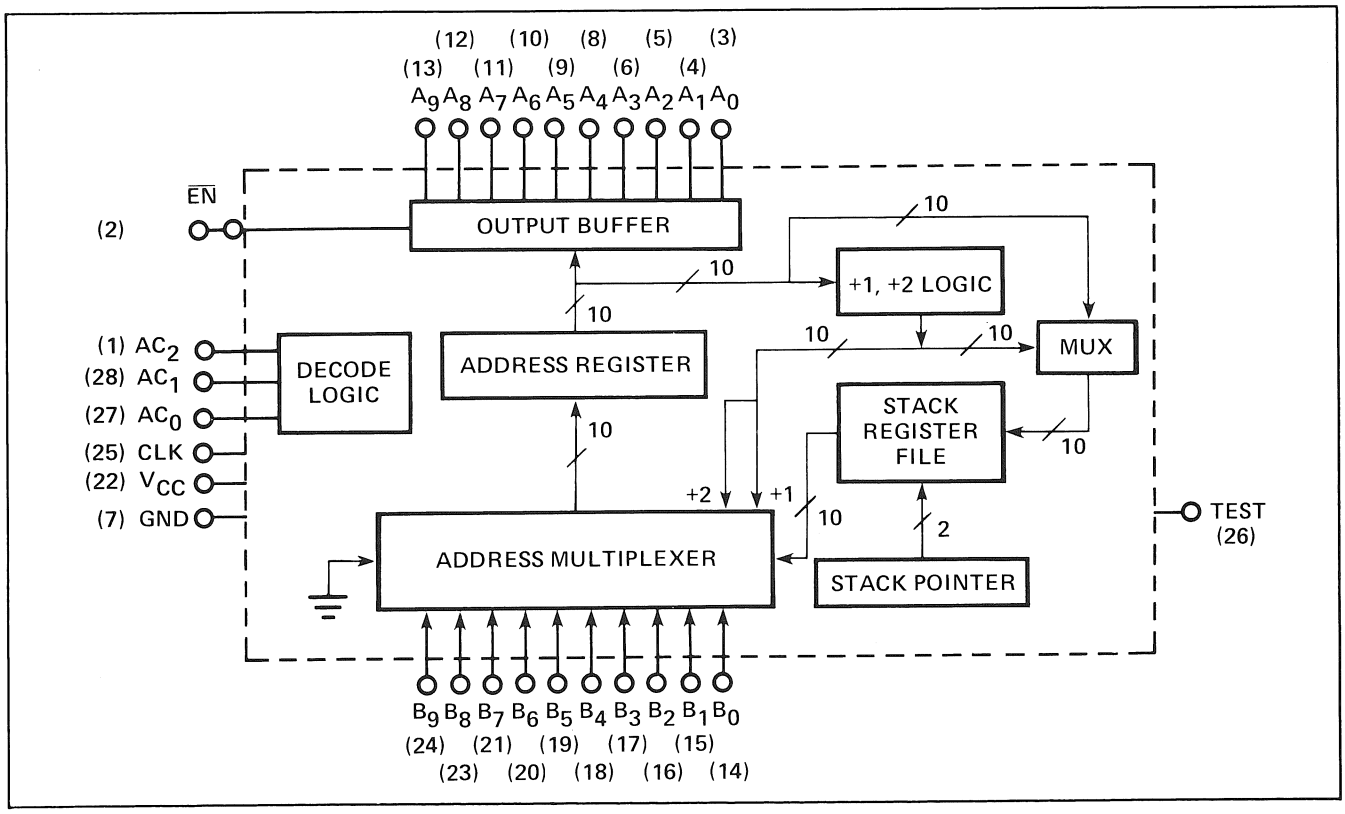


FIGURE 3.8

The 8X02's function is selected with a 3-bit control field called the Address Control Function input. Table 3.2 presents the mnemonic name and describes the operation of each Address Control Function.

TABLE 3.2

MNEMONIC	FUNCTION DESCRIPTION
TSK	AC_{2-0} = 000: TEST & SKIP Perform test on TEST INPUT LINE. If test is FALSE (LOW): Next Address = Current Address + 1 Stack Pointer unchanged If test is TRUE (HIGH): Next Address = Current Address + 2 (i.e. Skip next microinstruction) Stack Pointer unchanged
INC	AC_{2-0} = 001: INCREMENT Next Address = Current Address + 1 Stack Pointer unchanged
BLT	AC_{2-0} = -010: BRANCH TO LOOP IF TEST CONDITION TRUE. Perform test on TEST INPUT LINE. If test is FALSE (LOW): Next Address = Current Address + 1 Stack Pointer decremented by 1 If test is TRUE (HIGH): Next Address = Address from Stack Register File (POP) Stack Pointer decremented by 1
POP	AC_{2-0} = 011: POP STACK Next Address = Address from Stack Register File (POP) Stack Pointer decremented by 1
BSR	AC_{2-0} = 100: BRANCH TO SUBROUTINE IF TEST CONDITION TRUE. Perform test on TEST INPUT LINE. If test is FALSE (LOW): Next Address = Current Address + 1 Stack Pointer unchanged If test is TRUE (HIGH): Next Address = Branch Address Input (B_{0-9}) Stack Pointer incremented by 1 PUSH (write) Current Address + 1 → Stack Register File
PLP	AC_{2-0} = 101: PUSH FOR LOOPING Next Address = Current Address + 1 Stack Pointer incremented by 1 PUSH (write) Current Address → Stack Register File
BRT	AC_{2-0} = 110: BRANCH ON TEST CONDITION TRUE Perform test on TEST INPUT LINE. If test is FALSE (LOW): Next Address = Current Address + 1 Stack Pointer unchanged If test is TRUE (HIGH): Next Address = Branch Address Input (B_{0-9}) Stack Pointer unchanged
RST	AC_{2-0} = 111: RESET TO ZERO Next Address = 0 Stack Pointer unchanged

TABLE 3.3

Microp Mnemonic	Value Assigned to ACF
TSK	000B
INC	001B
BLT	010B
POP	011B
BSR	100B
PLP	101B
BRT	110B
RST	111B

The intrinsic microps for the 8X02 provide values for the Address Control Function (ACF) field. When the 8X02 intrinsic microps are used, a 3-bit field, ACF, must be defined in the Microinstruction Definition Section. The 8X02 microps make value assignments to the ACF field as specified in Table 3.3.

The 8X02 microps only affect the ACF input to the 8X02. Other control inputs such as Chip Enable and the Branch Address inputs must be assigned separately.

N3002 Microps

The N3002 Central Processing Element (CPE) is a data processing module designed as a 2-bit CPU slice. A CPU of 2n-bit word width can be constructed by combining n CPEs into a CPE array. To the microprogrammer, a 2n-bit CPE array is treated as a single data processing unit with twelve 2n-bit registers and a 2n-bit Arithmetic Logic Unit (ALU). A block diagram of a CPE array is presented in Figure 3.9

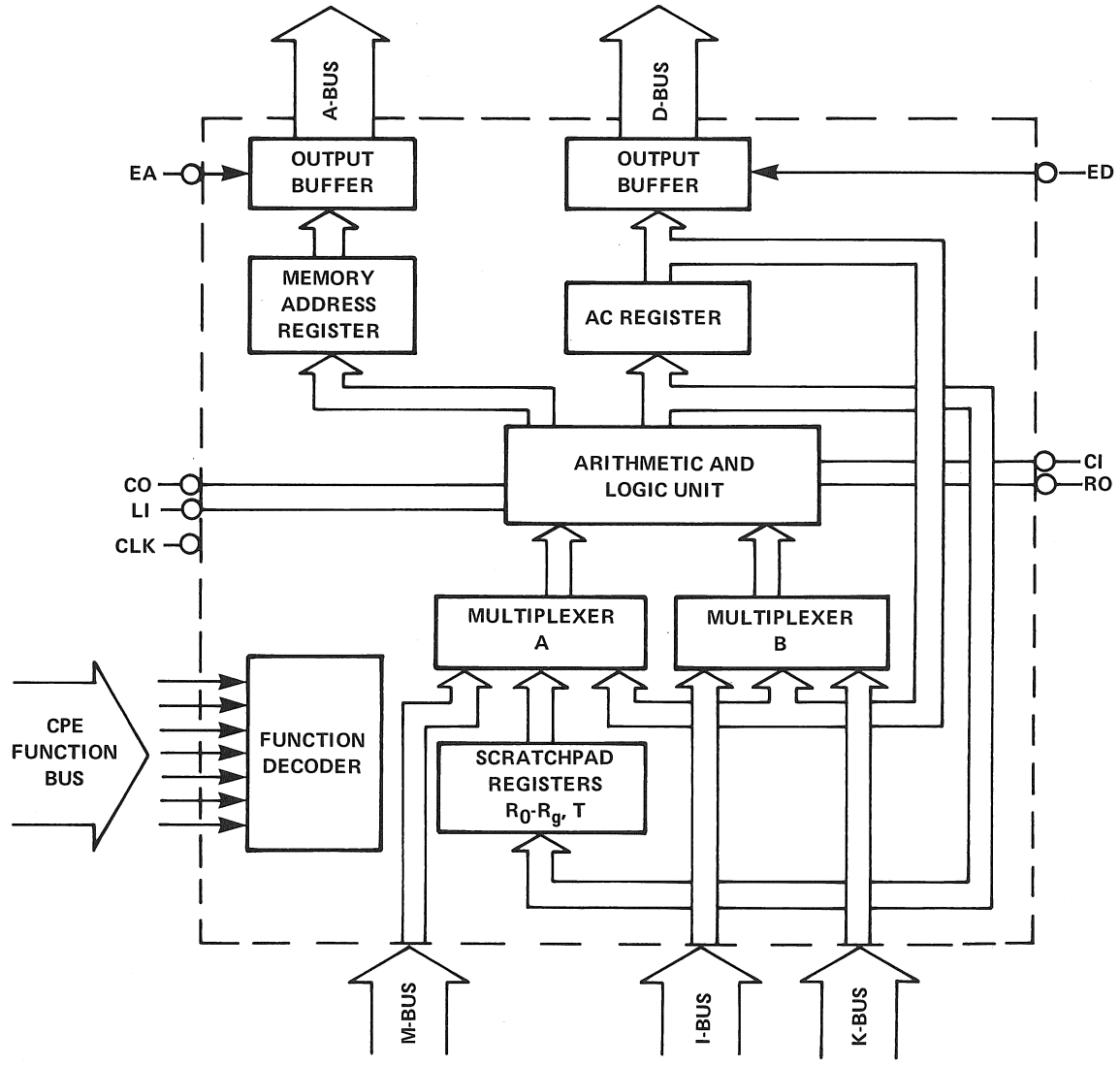


FIGURE 3.9

The function performed by the CPE array during any given microinstruction cycle is determined by a 7-bit control field presented to each CPE of the array by the CPE Function Bus. The N3002 intrinsic microps provide values for this 7-bit control field. Before the N3002 intrinsic microps are used, the microprogrammer must define a 7-bit field named F in the Microinstruction Definition Section.

The function performed by the CPE array (as determined by the F control field) can be modified by the K-Bus input to the array. Hence, some N3002 microps also provide a mask value for the K-Bus. The microprogrammer must therefore also define a K-Bus control field called KB in the Microinstruction Definition Section. The F field is divided into two groups: the F-Group and the R-Group. The F-Group consists of the three high order bits and specifies the general class of microfunction to be executed. The R-Group consists of the four low order bits and specifies the registers involved in the microfunction. Table 3.4 lists the binary values for the two groups.

The binary values listed in Table 3.4 are assigned to the control field F according to the intrinsic microp referenced in the Microinstruction Statement. Table 3.5 presents the general case N3002 microps. The general case microps make no assignments to the KB field.

TABLE 3.4

FUNCTION GROUP	F_6	F_5	F_4
0	0	0	0
1	0	0	1
2	0	1	0
3	0	1	1
4	1	0	0
5	1	0	1
6	1	1	0
7	1	1	1

REGISTER GROUP	REGISTER	F_3	F_2	F_1	F_0
I	R_0	0	0	0	0
	R_1	0	0	0	1
	R_2	0	0	1	0
	R_3	0	0	1	1
	R_4	0	1	0	0
	R_5	0	1	0	1
	R_6	0	1	1	0
	R_7	0	1	1	1
	R_8	1	0	0	0
	R_9	1	0	0	1
	T	1	1	0	0
	AC	1	1	0	1
II	T	1	0	1	0
	AC	1	0	1	1
III	T	1	1	1	0
	AC	1	1	1	1

TABLE 3.5

F-Group	R-Group	Microp	Micro-Function	
0	I	ARAMAC	$R_n + (AC \wedge K) + CI \rightarrow R_n$, AC	
	II	AMAM	$M + (AC \wedge K) + CI \rightarrow AT$	
1	I	MARK	$K \vee R_n \rightarrow MAR$	$R_n + CI + K \rightarrow R_n$
	II	MAMK	$K \vee M \rightarrow MAR$	$M + CI + K \rightarrow AT$
	III	ACAKAM	$(\overline{AT} \vee K) + (AT \wedge K) + CI \rightarrow AT$	
2	I	ASAM	$(AC \wedge K) - 1 + CI \rightarrow R_n$ (see Note 1)	
	II	ASAMX	$(AC \wedge K) - 1 + CI \rightarrow AT$	
	III	ASIM	$(I \wedge K) - 1 + CI \rightarrow AT$	
3	I	ARAM	$R_n + (AC \wedge K) + CI \rightarrow R_n$	
	II	AMAMX	$M + (AC \wedge K) + CI \rightarrow AT$	
	III	AAIM	$AT + (I \wedge K) + CI \rightarrow AT$	
4	I	NRAM	$CI \vee (R_n \wedge AC \wedge K) \rightarrow CO$	$R_n \wedge (AC \wedge K) \rightarrow R_n$
	II	NMAM	$CI \vee (M \wedge AC \wedge K) \rightarrow CO$	$M \wedge (AC \wedge K) \rightarrow AT$
	III	NAIM	$CI \vee (AT \wedge I \wedge K) \rightarrow CO$	$AT \wedge (I \wedge K) \rightarrow AT$
5	I	LTRM	$CI \vee (R_n \wedge K) \rightarrow CO$	$K \wedge R_n \rightarrow R_n$
	II	LTMM	$CI \vee (M \wedge K) \rightarrow CO$	$K \wedge M \rightarrow AT$
	III	LTAM	$CI \vee (AT \wedge K) \rightarrow CO$	$K \wedge AT \rightarrow AT$

TABLE 3.5 (cont.)

F-Group	R-Group	Microp	Micro-Function	
6	I	ORAM	$CI \vee (AC \wedge K) \rightarrow CO$	$R_n \vee (AC \wedge K) \rightarrow R_n$
	II	OMAM	$CI \vee (AC \wedge K) \rightarrow CO$	$M \vee (AC \wedge K) \rightarrow AT$
	III	OAIM	$CI \vee (I \wedge K) \rightarrow CO$	$AT \vee (I \wedge K) \rightarrow AT$
7	I	XRAM	$CI \vee (R_n \wedge AC \wedge K) \rightarrow CO$	$R_n \overline{\oplus} (AC \wedge K) \rightarrow R_n$
	II	XMAM	$CI \vee (M \wedge AC \wedge K) \rightarrow CO$	$M \overline{\oplus} (AC \wedge K) \rightarrow AT$
	III	XAIM	$CI \vee (AT \wedge I \wedge K) \rightarrow CO$	$AT \overline{\oplus} (I \wedge K) \rightarrow AT$

NOTE:

1. 2's complement arithmetic adds 111 . . . 11 to perform subtraction of 000 . . . 01.

Symbol	Meaning
I, K, M	Data on the I, K, and M busses, respectively
R_n	Contents of register n (R-Group I)
AC	Contents of the accumulator
AT	Contents of AC or T, as specified
CI	Data on the carry input
CO	Data on the carry output
L, H	As subscripts, designate low and high order bit, respectively
+	2's complement addition
-	2's complement subtraction
$\wedge$	Logical AND
$\vee$	Logical OR
$\overline{\oplus}$	Exclusive-NOR
$\rightarrow$	Deposit into

The microfunctions listed in Table 3.5 allow the microprogrammer to specify any mask value for the KB field. The general case microps are greatly simplified if the K-Bus value presented to the CPE array is forced to all zeros or all ones. The intrinsic N3002 microps include microps that assign all zeros or all ones to the KB field. These microps and the resulting microfunctions are listed in Table 3.6.

TABLE 3.6

F Field F-Group	F Field R-Group	KB Field	Microp	Functional Equation
0	I	all zeros	ILR	Rn + CI → Rn, AC
		all ones	ALR	AC + Rn + CI → Rn, AC
0	II	all zeros	AM	M + CI → AT
		all ones	AMA	M + AC + CI → AT
0	III	all zeros	SRA	$AT_L \rightarrow RO$ $AT_H \rightarrow AT_L$ $LI \rightarrow AT_H$
		all ones	–	
1	I	all zeros	MINR	Rn → MAR Rn + CI → Rn
		all ones	MARS	11 → MAR Rn – 1 + CI → Rn
1	II	all zeros	MAM	M → MAR Rn + CI → AT
		all ones	MAMS	11 → MAR M – 1 + CI → AT
1	III	all zeros	ACA	$\overline{AT}$ + CI → AT
		all ones	AAS	AT – 1 + CI → AT
2	I	all zeros	CSR	CI – 1 → Rn
		all ones	ASA	AC – 1 + CI → Rn
2	II	all zeros	–	
		all ones	–	
2	III	all zeros	–	
		all ones	ASI	I – 1 + CI → AT
3	I	all zeros	INR	Rn + CI → Rn
		all ones	ARA	AC + Rn + CI → Rn
3	II	all zeros	–	
		all ones	–	
3	III	all zeros	–	
		all ones	AAI	I + AT + CI → AT
4	I	all zeros	CLR	CI → CO 0 → Rn
		all ones	NRA	CI ∨ (Rn ∧ AC) → CO Rn ∧ AC → Rn
4	II	all zeros	–	
		all ones	NMA	CI ∨ (M ∧ AC) → CO M ∧ AC → AT
4	III	all zeros	–	
		all ones	NAI	CO ∨ (AT ∧ I) → CO AT∧I → AT

TABLE 3.6 (cont.)

F Field F-Group	F Field R-Group	KB Field	Microp	Functional Equation	
5	I	all zeros	–		
		all ones	LTR	CI ∨ Rn → CO	Rn → Rn
5	II	all zeros	–		
		all ones	LTM	CI ∨ M → CO	M → AT
5	III	all zeros	–		
		all ones	–		
6	I	all zeros	NOP	CI → CO	Rn → Rn
		all ones	ORA	CI ∨ AC → CO	Rn ∨ AC → Rn
6	II	all zeros	LM	CI → CO	M → AT
		all ones	OMA	CI ∨ AC → CO	M ∨ AC → AT
6	III	all zeros	–		
		all ones	OAI	CI ∨ I → CO	I ∨ AT → AT
7	I	all zeros	LCR	CI → CO	$\overline{Rn}$ → Rn
		all ones	XRA	CI ∨ (Rn ∧ AC) → CO	Rn $\bar{\oplus}$ AC → AT
7	II	all zeros	LCM	CI → CO	$\overline{M}$ → AT
		all ones	XMA	CI ∨ (M ∧ AC) → CO	M $\bar{\oplus}$ AC → AT
7	III	all zeros	–		
		all ones	XAI	CI ∨ (AT ∧ I) → CO	I $\bar{\oplus}$ AT → AT

NOTE:

1. 2's complement arithmetic adds 111 . . . 11 to perform subtraction of 000 . . . 01.

Symbol	Meaning
I, K, M	Data on the I, K, and M busses, respectively
R_n	Contents of register n (R-Group I)
AC	Contents of the accumulator
AT	Contents of AC or T, as specified
CI	Data on the carry input
CO	Data on the carry output
L, H	As subscripts, designate low and high order bit, respectively
+	2's complement addition
–	2's complement subtraction

TABLE 3.6 (cont.)

Symbol	Meaning
$\wedge$	Logical AND
$\vee$	Logical OR
$\overline{\oplus}$	Exclusive-NOR
$\rightarrow$	Deposit into

TABLE 3.7

Register Name	Value
R0	0000B
R1	0001B
R2	0010B
R3	0011B
R4	0100B
R5	0101B
R6	0110B
R7	0111B
R8	1000B
R9	1001B
T	1010B
AC	1011B

Register names — R0 through R9 may only be used with microps from R Group I. Register names T and AC may be used with any 3002 microp.

The N3002 microps require a single argument enclosed in parentheses. This argument specifies the register to be used in the microfunction. The argument may be an expression, but intrinsic value symbols are provided when the N3002 intrinsic microps are used. The N3002 value symbols represent register names, as defined in Table 3.7. The actual value assigned to $F_{(0\text{-}3)}$ is determined by the R-Group of the microp.

Table 3.8 presents some examples of N3002 microps, their resulting functions and the F field and KB field values they produce.

N2901-1 Microps

The N2901-1 is a high speed bipolar 4-bit RALU (Register/Arithmetic Logic Unit) slice. The N2901-1 is microprogrammable and the Micro Assembler intrinsic microps simplify the task of programming the device.

TABLE 3.8

Microp Specification	Functional Equation(s)	F Field Value	KB Field Value
ARAM(R6)	R6 + (AC $\wedge$ K) + CI $\rightarrow$ R6	011 0110B	default
LTAM(T)	CI $\vee$ (T $\wedge$ K) $\rightarrow$ CO K $\wedge$ T $\rightarrow$ T	101 1110B	default
NOP(R0)	CI $\rightarrow$ CO R0 $\rightarrow$ R0	110 0000B	all zeros
XRA(R5)	CI $\vee$ (R5 $\wedge$ AC) $\rightarrow$ CO R5 $\overline{\oplus}$ AC $\rightarrow$ R5	111 0101B	all ones

Basic Architecture

The N2901-1 consists of a high speed ALU, a 16 register dual port RAM, a 4-bit Q Register and a powerful data routing structure for moving data between the ALU and the internal registers. The basic organization of these circuit elements is illustrated in the block diagram of the N2901-1 presented in Figure 3.10.

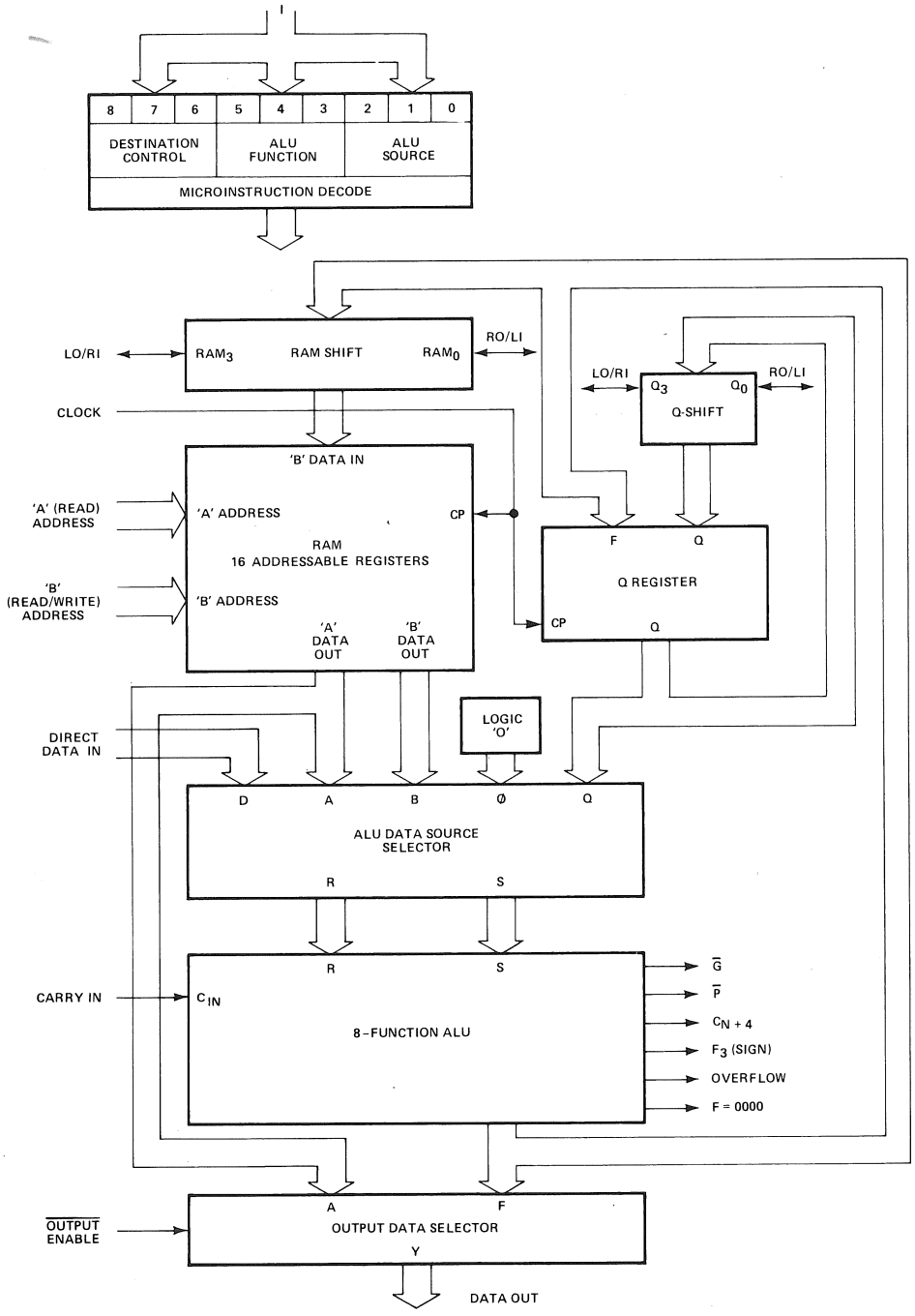


FIGURE 3.10

Dual Port RAM – The N2901-1 contains a 16-word by 4-bit dual port register array. The two ports, the A-port and B-port, are controlled by two 4-bit address fields input to the device. These two input fields, the A-address and B-address, are control fields generated by the N2901-1 microps. Data in any register can be accessed by either port. The two ports operate independently, and hence if the same value is input to the two address fields, the same data will appear at both ports. Data written into the RAM is always written into the address defined by the B-Address input field. The two address fields must be defined before the N2901-1 intrinsic microps are selected. The field names are A and B.

ALU – The high speed ALU performs three binary arithmetic and five logic functions on two source operands. The two source operands are called the R-operand and the S-operand. The R-operand is driven by a two input multiplexer that selects either the 4-bit data field (direct data input to the device), or the A-port of the register array. The S-operand is driven by a three input multiplexer that selects either the A-port of the register array, the B-port of the register array, or the Q-register. Both R + S inputs can be forced to all zeros.

Shift Logic – The input to the 16 register RAM is driven by a three input multiplexer. This multiplexer provides a means of shifting data output from the ALU before it is written into the RAM. A similar shift multiplexer drives the Q-register input. The shift multiplexers provide shift-left, shift-right and no-shift operations on data written into the RAM and the Q-register. Separate bi-directional pins, RAM_0, RAM_3, Q_0 and Q_3 are provided for shifting data between N2901-1s.

Data Out (Y-outputs) – Data out of the device appears on the Y-outputs. The Y-outputs are driven by a two input multiplexer that selects either the A-port or the ALU results.

The preceding functional blocks are more clearly illustrated in the detailed block diagram of the N2901, presented in Figure 3.12.

In addition to the two RAM address fields, the N2901-1 microps also generate a 9-bit Microfunction input field for the RALU. The Microfunction field (I_{0-8}), determines the function that will be performed during a microcycle (the time required to execute one microinstruction). It specifies the source operands, the destination of the ALU results and the shift operation performed on the ALU results prior to storage in the destination register. The Microfunction input field is subdivided into three 3-bit subfields that must be defined with FIELD Statements before the N2901 microps are called with an INTRINSIC Statement. The Microfunction subfields are listed below:

Control Bits	Field Name	Field Function
$I_{(0-2)}$	SRC (Source)	Determines the source operands.
$I_{(3-5)}$	FUN (Function)	Determines the operation to be performed.
$I_{(6-8)}$	DST (Destination)	Determines the destination of the ALU data output and the shift operation performed on the data prior to storage.

The Microps

The N2901-1 microps are included in the Microinstruction Definition Section with an INTRINSIC Statement. Once the N2901-1 microps have been included in the Microinstruction Definition Section, they may be referenced in Microinstruction Statements within the Program Section. The intrinsic N2901-1 microps may be located anywhere within a Microinstruction Statement. The basic format for the N2901-1 microps is illustrated in Figure 3.11.

OP-CODE source_operand [destination_operand]

FIGURE 3.11

Registers assigned in the source and destination operands have been predefined in the intrinsic N2901 microps with the following symbols:

R0, R1, R2, R3, R4, R5, R6, R7,
R8, R9, R10, R11, R12, R13, R14, R15, and Q

R0 through R15 specify RAM addresses accessed via the A-address and B-address input fields to the N2901. Q specifies the Q-register.

The N2901-1 microps will be presented in two sections. First, the source operands and the Op-Codes will be presented by functional groups, and then the destination operand will be described.

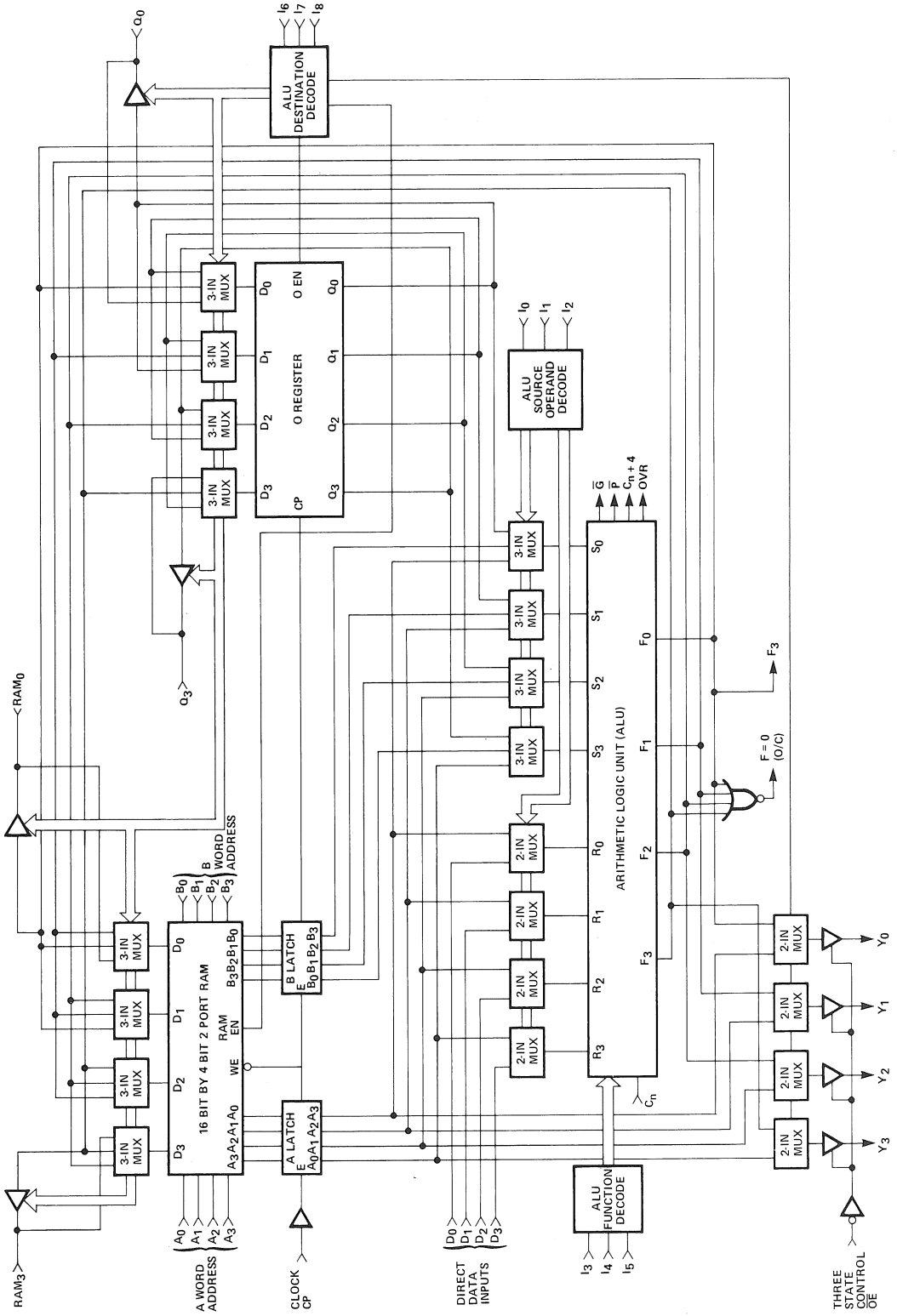

ALU DESTINATION DECODE
O REGISTER
ALU SOURCE OPERAND DECODE
ARITHMETIC LOGIC UNIT (ALU)
16 BIT BY 4 BIT 2 PORT RAM
B LATCH
A LATCH
ALU FUNCTION DECODE
3-IN MUX
2-IN MUX
WORD ADDRESS
A WORD ADDRESS
CLOCK CP
DIRECT DATA INPUTS
THREE STATE CONTROL OE
F = 0 (O/C)
OVR

FIGURE 3.12

The Op-Code and Source Operands

The Op-Codes are divided into four groups. The groups are organized by the possible source operands. Tables 3.9 and 3.10 list the Op-Code mnemonics and define their functions.

TABLE 3.9

GROUP I	
Mnemonic	**Function**
AR	Add Register
SR	Subtract Register
RSR	Reverse Subtract Register
ORR	OR Register
NR	AND Register
NCR	AND Complemented Register
XR	Exclusive OR Register
XCR	Exclusive OR Complemented Register

TABLE 3.10

GROUP II	
Mnemonic	**Function**
IR	Increment Register
DR	Decrement Register
LNR	Load Negative Register
LR	Load Register
LCR	Load Complemented Register
GROUP III	
AD	Add Data-in
SD	Subtract Data-in
RSD	Reverse Subtract Data-in
OD	OR Data-in
ND	AND Data-in
NCD	AND Complemented Data-in
XD	Exclusive OR Data-in
XCR	Exclusive OR Complemented Data in
GROUP IV	
ID	Increment Data-in
LND	Load Negative Data-in
DD	Decrement Data-in
LD	Load Data-in
LZ	Load Zero
LCD	Load Complemented Data-in

Figure 3.13 presents the general format for Group I microps.

OP-CODE (r,s) [destination_operand]

FIGURE 3.13

The source operand for Group I requires that the source for both the r and s inputs to the ALU be specified. For this group, the source for the r input is always the A-port, and hence, the r value must be a register number. The source for the s input can be either the B-port or the Q-register. A register number specifies the B-port and Q specifies the Q-register. Table 3.11 lists the Group I Op-Codes, their respective functions and the resulting SRC and FUN field values.

TABLE 3.11

Group I Microps

FUN Value $I_{(3,5)}$ / SRC Value $I_{(0,2)}$		0	1
Op-Code / (r,s)		(A, Q)	(A, B)
0	AR	A + Q + CI	A + B + CI
1	SR	Q – A – CI	B – A – CI
2	RSR	A – Q – CI	A – B – CI
3	ORR	$A \vee Q$	$A \vee B$
4	NR	$A \wedge Q$	$A \wedge B$
5	NCR	$\overline{A} \wedge Q$	$\overline{A} \wedge B$
6	XR	$A \forall Q$	$A \forall B$
7	XCR	$\overline{A} \forall Q$	$\overline{A} \forall B$

*Notes: $\vee$ = OR
$\wedge$ = AND
$\forall$ = Exclusive OR
CI = Carry In
SRC and FUN value given in octal

The general format of Group II microps is presented in Figure 3.14.

OP-CODE(s) [destination_operand]

FIGURE 3.14

The source operand for Group II microps requires that only one source register be specified. For the Group II microps the r input to the ALU is forced to all zeros. The register specified is therefore the s input to the ALU. RAM data (R0-R15) or the Q-register may be specified as the source operand. Table 3.12 lists the group II Op-Codes, their respective functions and the resulting SRC and FUN field values.

TABLE 3.12

Group II Microps				
FUN Value $I_{(3,5)}$	SRC Value $I_{(0,2)}$ / (r,s) / Op-Code	2	3	4
		0, Q	0, B	0, A
0	IR	Q + CI	B + CI	A + CI
1	DR	Q – CI	B – CI	A – CI
2	LNR	–Q – CI	–B – CI	–A – CI
3	LR	Q	B	A
4	–	–	–	–
5	–	–	–	–
6	–	–	–	–
7	LCR	$\overline{Q}$	$\overline{B}$	$\overline{A}$

*Notes: CI = Carry In
SRC and FUN values given in octal.

Figure 3.15 presents the general format for the Group III microps.

OP-CODE(s) [destination_operand]

FIGURE 3.15

Like Group II, Group III requires only one source operand. In this case, however, the r input to the ALU is driven by the data-in inputs. The source operand specifies the s input to the ALU. This may be RAM data accessed by the A-port or the Q-register. Table 3.13 lists the Group III Op-Codes, their functions and the resulting SRC and FUN field values.

TABLE 3.13

Group III Microps			
FUN Value $I_{(3,5)}$	SRC Value $I_{(0,2)}$ / (r,s) / Op-Code	5	6
		D, A	D, Q
0	AD	D + A + CI	D + Q + CI
1	SD	A – D – CI	Q – D – CI
2	RSD	D – A – CI	D – Q – CI
3	OD	D $\vee$ A	D $\vee$ Q
4	ND	D $\wedge$ A	D $\wedge$ Q
5	NCD	$\overline{D}\wedge$ A	$\overline{D}\wedge$ Q
6	XD	D $\forall$ A	D $\forall$ Q
7	XCD	$\overline{D}\forall$ A	$\overline{D}\forall$ Q

*Notes: $\vee$ = OR $\wedge$ = AND $\forall$ = Exclusive OR
CI = Carry In
SCR and FUN values given in octal.

Figure 3.16 presents Group IV's format.

OP-CODE [destination_operand]

FIGURE 3.16

Group IV microps require no source operand. The r input to the ALU is driven by the data-in inputs and the s input is forced to all zeros. Table 3.14 lists the Group IV Op-codes, their functions and the resulting SRC and FUN field values.

TABLE 3.14

Group IV Microps		
FUN Value I(3,5)	SRC Value I(0,2)	7
	(r,s) Op-Code	D, 0
0	ID	D + CI
1	LND	−D − CI
2	DD	D − CI
3	LD	D
4	LZ	0
5	—	—
6	—	—
7	LCD	$\overline{D}$

*Notes: CI = Carry In
SRC and FUN values are given in octal

The Destination Operand

The results of the various Op-Code Groups are stored in RAM or in the Q-register specified by the destination operand. The destination operand also specifies the shift operation performed on the ALU output prior to its being stored in the destination register. The format of the destination operand is illustrated in Figure 3.17.

store_function [(destination_register)]

FIGURE 3.17

The store function is specified by the store function mnemonics. These mnemonics and their full names are given in Table 3.15.

TABLE 3.15

Mnemonic	Store Function
STR	STore in RAM
STQ	STore in Q-register
SLR	Shift Left and store in RAM
SLRQ	Shift Left and store in RAM and Q-register
SRR	Shift Right and store in RAM
SRRQ	Shift Right and store in RAM and Q-register

When the destination register specified is a RAM location, the ALU results are stored in the RAM location pointed to by the B-address. If the B-port address is specified by the source operand, the destination register portion of the destination operand may be omitted. (If the B-port address is specified in the source operand, and a different B-port address is specified by the destination operand, the source operand B-port address is used. No error indication is given.)

A special case of the destination operand is when the data outputs (Y-Outputs) of the N2901-1 are driven by the A-port. (Normally the Y-Outputs are driven by the ALU results.) This special case is implemented with the STR store function. The format of this operation is presented in Figure 3.18.

STR [(destination-register)] , Y [(A-port-address)]

FIGURE 3.18

The A-port address need only be included in the destination operand if no A-port address is specified in the source operand. Note the use of a comma as a delimiter. Figure 3.19 summarizes the possible forms of the destination operand.

{ STR, STQ, SLR, SLRQ, SRR, SRRQ }	[(b)]
{ STR }	[(b)] [,Y[(a)]]

*Note: b=B-port address
a=A-port address

FIGURE 3.19

The entire destination operand is enclosed in brackets because it may be omitted. The default for the destination operand is a NOP (No-OPeration; in this case, no data is stored).

Table 3.16 lists the store functions and their resulting DST field values.

Note the RAM Shifter and Q Shifter columns in Table 3.16. They describe the interaction between the store function and the cascade shift pins. RAM_0, RAM_3, Q_0 and Q_3 are bidirectional signal lines that facilitate shifts between two or more N2901s. Figure 3.20 presents some representative microps and the operations they produce.

AR(R2, R7) STR	R2 + R7 + CI → R7, Y
SR(R9,Q) STQ	Q – R9 + CI → Q, Y
XCR(R0,Q) STR(R11)	$\overline{R0} \forall Q \rightarrow R11$, Y
IR(R6) STR(R13), Y	R6 + CI → R13, R6 → Y
LCD STQ,Y(R1)	$\overline{D} \rightarrow Q$, R1 → Y
ND(R14) SRR(R6)	D∧ R14→Y, (D∧R14)/2→ R6

FIGURE 3.20

TABLE 3.16

Store Function	Micro Code			RAM Function		Q Register Function		Y Output	RAM Shifter		Q Shifter	
	I_8 I_7 I_6		Octal Code	Shift	Load	Shift	Load		RAM_0 LO/RI	RAM_3 LI/RO	Q_0 LO/RI	Q_3 LI/RO
STQ	L	L L	0	X	None	None	F→Q	F	X	X	X	X
	L	L H	1	X	None	X	None	F	X	X	X	X
STR, Y	L	H L	2	None	F→B	X	None	A	X	X	X	X
STR	L	H H	3	None	F→B	X	None	F	X	X	X	X
SRRQ	H	L L	4	Right (Down)	F/2→B	Right (Down)	Q/2→Q	F	F_0	IN_3	Q_0	IN_3
SRR	H	L H	5	Right (Down)	F/2→B	X	None	F	F_0	IN_3	Q_0	X
SLRQ	H	H L	6	Left (Up)	2F→B	Left (Up)	2Q→Q	F	IN_0	F_3	IN_0	Q_3
SLR	H	H H	7	Left (Up)	2F→B	X	None	F	IN_0	F_3	X	Q_3

X = Don't care Electrically, the shift pin is a TTL input internally connected to a three-state output which is in the high-impedance state.

PART FOUR – THE MICRO ASSEMBLER'S OUTPUT

As the Micro Assembler assembles the source, it produces two output files. These are the listing file and the object file. The listing file can be used to completely document a microprogram as it contains both source and object information. The object file is the Intermediate Object Text to be input to the Micro Format Program.

THE LISTING FILE

The listing file is generated in two different sections. The first section is the source/object listing and the second is the cross reference listing.

The listing file is formatted for hard copy terminals and printers. The listings are formatted in 80 character lines grouped into pages. Each page begins with a page heading. Output to the listing file can be controlled with the listing directive statements embedded in the source.

The page heading begins with the following record:

SIGNETICS BIPOLAR MICRO ASSEMBLER (A)
********** PAGE 00000

The asterisks indicate user text defined with a TITLE statement. The TITLE statement defines a 28 character title heading. The page heading contains two lines, the second depending on which listing is being produced.

Source/Object Listing

This section of the listing has two types of records, the formatted source and the formatted object. The formatted source listing records consist of a source line number followed by the first 72 positions of a source line. The source line number is a count of source lines maintained by the micro assembly program.

The formatted object listing records consist of the memory block address of the data (in hexadecimal) followed by the object data in binary with blanks between microinstruction fields. If the object data is from a DCL statement, blanks are placed between successive memory block words. As required, the object fields or words may be continued on successive object listing records. The object listing records follow the source listing records of the statement which produced the object data. Source statements which produce no object data (such as directive statements) are not followed by object listing records. Error messages, should there be any, are listed below the line they reference.

The second line in the source/object heading is a list of the microinstruction's Fields. In the event of multiple micro-instructions, the first one defined provides the field names for the heading.

An Example of Source/Object Listing

```
SIGNETICS BIPOLAR MICRO ASSEMBLER(A)   EXPERIMENTAL DISK CONTROLLER  PAGE 00002
  ADDR:     F        K8     ACF       A

00032         PROGRAM DISK_CONTROLLER WIDTH 28 LENGTH 512;
00033         ORG 3DH;
00034  GO:    LM(AC) BLT "SAVE M BUS" ;
  003D: 1101011 00000000 010 0000111110
00035         AAI(T) BRT A=GO;
  003E: 0111110 11111111 110 0000111101
00036         ALR(R9) BRT A=FETCH;
     *****  SYMBOL ERROR 01  *****     FETCH
  003F: 0001001 11111111 110 0000000000
00037         ILR(AC);
  0040: 0001101 00000000 001 0001000001
```

FIGURE 4.1

Cross Reference Listing

This section of the listing file is an alphabetic list of all symbols (except reserved words) defined in the assembly. The second heading line labels information in the listing. For each symbol the following information is listed:

+ Symbol type, a single letter code defining the type of symbol; as follows,

  D – Label of Microinstruction or DCL statement.

  E – Label of EQU statement

  S – Label of SET statement.

  P – Program symbol, memory block name.

  F – Field name.

  M – Microp name.

  A – Microp argument name.

+ Symbol value, if type is D, E or S, in hexadecimal.
+ Symbol name.
+ Source line number of definition.
+ A list of source line numbers of references to the symbol.

An Example of a Cross Reference Listing

NAME OF REFERENCING MICROP

```
SIGNETICS BIPOLAR MICRO ASSEMBLER(4)    EXPERIMENTAL DISK CONTRCLLER   PAGE 00004
TYPE VALUE        SYMBOL                        REFERENCES

A    *@RGI*       @OP     } OP USED AS ARGUMENT IN   CCC20* 00021  00021
A    *@RGII*      @OP     } THREE DIFFERENT MICROPS  CCC22* 00022
A    *@RGIII*     @OP     }                          CCC23* 00024
A    *@PGI*       @P                                 CCC20* 00020  00020  00021  00021  00021
A    *@RGII*      @R                                 CCC22* 00022  00022  00022
A    *@RGIII*     @P                                 CCC23* 00023  00023  00024
A    * ARAMAC*    @R                                 CCC25* 00025
A    * ILR*       @R                                 CCC26* 00026
A    * AIR*       @R                                 CCC27* 00027
A    *LM*         @R                                 CCC28* 00028
A    * AAI*       @R                                 CCC29* 00029
M                 @PGI                               CCC20* 00025  00026  00027
M                 @RGII                              CCC22* 00028
M                 @RGIII                             CCC23* 00029
F      0010       A        DECIMAL                   CCC06* 00035  00036
M                 AAI      VALUE                     CCC29* 00035
E    000B         AC       INDENTED                  CCC19* 00020  00022  00023  00034  00037
F      0003       ACF                                CCC05* 00030  00031
M                 ALR                                CCC27* 00036
M                 ARAMAC                             CCC25*
M                 BLT                                CCC30* 00034
M                 BRT                                CCC31* 00035  00036
P      0028       DISKCONTRCLLER                     CCC32*
F      0007       F                                  CCC03* 00021  00021  00022  00024
U                 FETCH                              CCC36
D    003D         GO                                 CCC34* 00035
M                 ILR                                CCC26* 00037
F      0008       KP                                 CCC04* 00026  00027  00028  00029
M                 LM                                 CCC28* 00034
F    0000         R0                                 CCC08*
F    0001         R1                                 CCC09*
F    0002         R2                                 CCC10*
F    0003         R3                                 CCC11*   ASTERISK DENOTES LINE
F    0004         R4                                 CCC12*<- WHERE SYMBOL IS DEFINED
F    0005         R5                                 CCC13*
F    0006         R6                                 CCC14*
F    0007         R7                                 CCC15*
F    0008         R8                                 CCC16*
F    0009         R9                                 CCC17* 00036
F    000A         T                                  CCC18* 00021  00022  00023  00035

*END OF ASSEMBLY*  SOURCE LINES CCC39  ERROR LINES 00001  OBJECT LINES 00002
```

NOTE: VALUE FOR FIELD NAMES AND PROGRAM NAME SPECIFIES WIDTH.

FIGURE 4.2

INTERMEDIATE OBJECT TEXT

The micro assembly program produces an object file that is used as input to the micro format program. This object file, called the Intermediate Object Text, consists of object data and object format information. This information is formatted into a set of logical object records; each logical record is separated by a semicolon. The first data character of each logical record defines the type of record. The following types are used:

F – Microinstruction format information.

P – Program block information.

I – Microinstruction object data.

D – Object data from DCL statements.

E – Object file terminator.

The object file organization follows the source input organization. The F records are produced by the definition block and precede all other records. Each program block in the source input produces a P record followed by multiple D and I records. The logical object records express *all* numeric values in hexadecimal format.

F Record Format

The first F record defines the microinstruction format and the following F records define fields in the microinstruction format. The first F record has the following format:

Fnnggg;

where nn is the format number and ggg is the microinstruction width.

For the basic micro assembly language, nn is a constant "01". The remaining F records define fields and have the following format:

Fnndddgggsss. . .;

where nn is the format number. ddd and ggg are the displacement and width respectively of the field. sss. . . is the field name. The field records follow the order of field definition in the source definition block.

P Record Format

The P record heads a program block in the object file. It has the following format:

Pwwwbbbsss. . .;

where www is the word size of the program block in bits and bbb is the number of bits required to address this program block. sss. . . is the name of the program block.

I Record Format

The I record contains the object data for one microinstruction and has the following format:

Innaaaattt . . . ;

where nn is the format number and aaaa is the address in the program block for this microinstruction. aaaa is variable length. The length of aaaa in hexadecimal digits is the minimum number of digits required to represent an address in the current memory module. The number of digits in aaaa is equal to the integer portion of (bbb + 3)/4 where bbb is from the last P card. tttt. . . is the object data text.

The object text (ttt. . .) consists of a list of numeric values without separators. Each numeric value corresponds to a field in the microinstruction format. The field values are in the same order as the F records. The number of hexadecimal digits in each value is determined by the field length. The number of digits in a field value is equal to the integer portion of (ggg + 3)/4 where ggg is from the appropriate F record. There are no separators between the field values; the first digit of a field value immediately follows the last digit of the previous field value in ttt. . .

D Record Format

The D record contains the object data from a source DCL statement and has the following format;

Daaaaggggtt. . .;

where aaaa is the address in the memory block for the DCL data and gggg is the number of bits in the data value. aaaa has the same format as in the I record, and gggg is always four hexadecimal digits. ttt. . . is the object data text. The number of hexadecimal digits in ttt. . . is equal to the integer portion of (gggg + 3)/4.

E Record Format

The E record terminates the micro assembly program object text and has the following format:

Eaaaa;

where aaaa is the execution address and may be omitted. The optional expression in the source input terminating END statement supplies the value for aaaa.

An Example of Intermediate Object Text

```
F0101C;F01015007F;F0100D008K5;F0100A003ACF;F0100000AA;P01C009DISKCONTROLLER;I01
003D6B002000;I01003E3EFF603D;I01003F09FF6000;I0100400D000000;E;
```

FIGURE 4.3

PART FIVE – THE MICRO FORMAT PROGRAM

THE MICRO FORMAT PROGRAM – AN OVERVIEW

The final phase of microprogram development is to take the microprogram (as assembled into intermediate object text), and place it in Firmware. This task is accomplished by the Micro Format Program. The Micro Format Program takes the intermediate object text and partitions the microprogram into PROM data fields. The PROM data fields are then formatted onto paper tapes that can be read by PROM programmers. Once the microprogrammer has obtained PROM programming tapes, the job of the Micro Assembler is finished and the microprogrammer is well on his way to delivering a microprogrammed system.

The Micro Format Program requires as input the intermediate object text and a command file of statements that describe the Control Store PROMs, assign microinstruction bits to the PROMs, and describe the format of the output paper tapes. The microinstruction bits may be arbitrarily assigned to the Control Store PROMs. This feature allows the microprogrammer to alter the Field-PROM assignments for PCB layout convenience. Bits assigned to PROMs may be inverted for negative true logic implementations. The Micro Format Program supports three paper tape formats intrinsically. Optionally, the microprogrammer may specify other paper tape formats.

The Micro Format Program also generates a listing. The listing is produced in two sections. The first section lists the command file and any errors encountered. The second section provides a summary listing of the microinstruction bit assignments to PROM modules.

PLACING THE MICROPROGRAM IN PROMs

The Micro Format Program divides the microprogram into PROM modules under direction of the command file. The command file is built with Micro Format statements. These statements are constructed from the same basic language elements and syntax as the Micro Assembly Program.

The command file consists of groups of statements that completely describe each PROM module. Each group contains three kinds of statements. These are:

1. The MEMORY Statement which specifies the size of the PROM module (e.g., 512 X 8 is the size of the 82S115).
2. The OUTPUT Statement which directs the Micro Format Program to output a section of the intermediate object text into a PROM module.
3. The SELECT Statement which specifies which Fields or bits of Fields are to be placed in the PROM module.

Each PROM module requires a combination of the above statements.

The MEMORY Statement

The format of the MEMORY statement is presented in Figure 5.1.

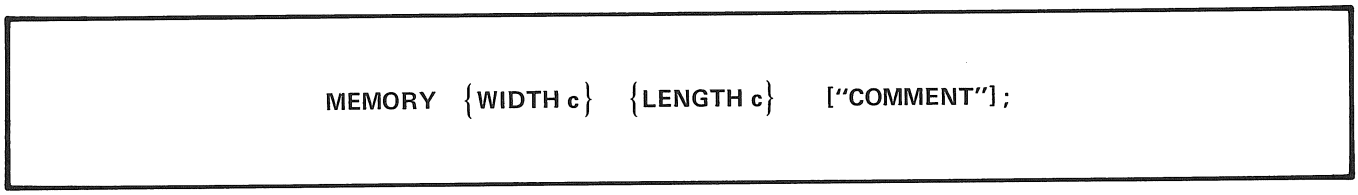


FIGURE 5.1

The statement consists of the MEMORY opcode followed by the memory word width specification and a number of memory words specification. The PROM dimensions specified by the MEMORY statement are used for subsequent object output by OUTPUT statements. The initial PROM dimensions are a word width of 8 bits and a memory size of 512 words.

The width specification consists of the keyword – WIDTH followed by a numeric constant. It must immediately follow the MEMORY opcode. The numeric constant specifies the memory word width in bits.

The memory size specification consists of the keyword – LENGTH followed by a numeric constant. The length operand must immediately follow the width specification. The numeric constant is the number of words in a PROM module (memory length).

Examples: MEMORY Statements.

```
MEMORY WIDTH 8 LENGTH 512
  "MICROINSTRUCTION PROM" ;

MEMORY WIDTH 4 LENGTH 256
  "82S226 256x4 ROM" ;
```

The OUTPUT Statement

The OUTPUT Statement is used in conjunction with a MEMORY Statement to create a PROM module. The OUTPUT Statement selects the Program Section from which the PROM module (defined by the MEMORY Statement) will be taken. This might be a portion of the microprogram or a control PROM built entirely from DCL Statements. If the OUTPUT Statement is subdividing the microprogram, its FROM and TO operands select the vertical boundaries of the microprogram that is to be placed into the PROM module (e.g., from 0 to 255 of a microprogram containing 512 instructions). The format of the OUTPUT Statement is given in Figure 5.2.

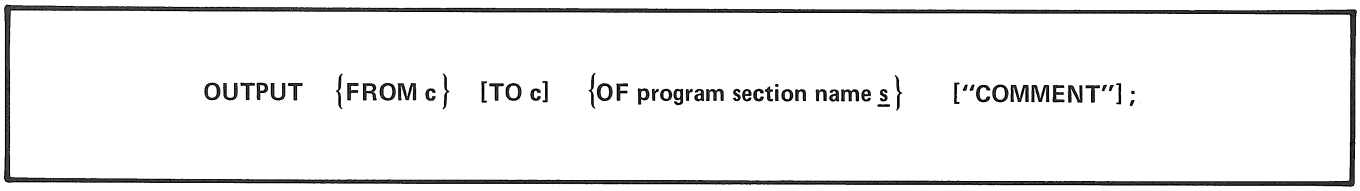


FIGURE 5.2

The FROM operand selects the beginning address of the intermediate object Program Section for output. The operand consists of the FROM keyword followed by a numeric constant. The numeric constant becomes the beginning address value. When the intermediate object is processed to produce the PROM module, only Program Section object data with addresses equal or greater than this value are selected for output. Addresses of a PROM module always begin at address zero, and the intermediate object addresses are adjusted to relative zero. Object data with an address equal to the from value is placed at PROM module address zero, object data with an address equal to the from value plus one is placed at PROM module address one, and so forth. The FROM operand must immediately follow the OUTPUT opcode and may not be omitted.

The TO operand selects the ending address of the Program Section object data that will be placed in the PROM module. The TO operand consists of the keyword TO and a numeric constant. The numeric constant is the address of the last microinstruction or DCL data word to be placed in the PROM module. The address area selected by the FROM and TO operands must not be larger than the PROM module defined by the MEMORY Statement.

The assignment of PROM addresses to individual PROM words need not be in ascending order. If the FROM value is greater than the TO value, address assignments will be made in descending order.

The TO operand may be omitted. If the TO operand is omitted, the default TO value is calculated by adding the PROM module length minus one to the FROM operand value.

The next language element in the OUTPUT Statement is the Program Section specification operand. The Program Section operand selects which of the assembled Program Sections will provide the object data for the PROM module. The specification consists of the keyword OF and a symbol. The symbol must be the name of a Program Section in the intermediate object text output by the Micro Assembly Program. Program Sections were named with PROGRAM statements in the Micro Assembly source.

The SELECT Statement

If a microprogram is over 8 bits wide, it will require more than one PROM for its Control Store firmware. This requires that the microinstruction's Fields be subdivided into individual bits before they can be assigned to multiple PROMs. The selection of which bits are to be placed in which PROM is made with a SELECT Statement. The format of the SELECT statement is illustrated in Figure 5.3.

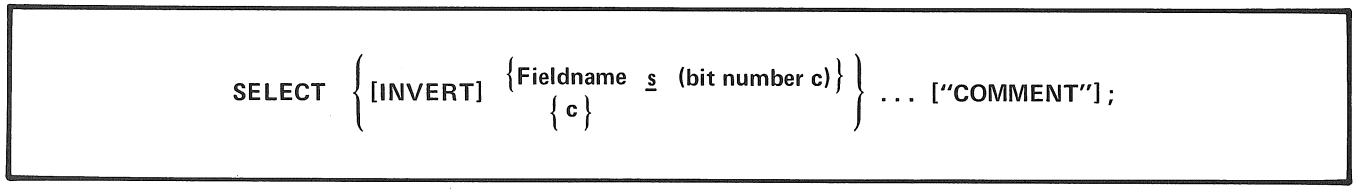


FIGURE 5.3

The SELECT statement consists of the SELECT opcode followed by a Field operand list. The Field operand list describes how the microinstruction fields are formatted into words in the PROM module. The PROM module to which the SELECT statement applies must be requested with a *preceding* OUTPUT statement. No statements may be placed between a SELECT statement and its associated OUTPUT statement except other SELECT statements.

The field operand list is a sequence of Field operands which select bits from microinstruction Fields. A Field operand consists of a symbol followed by a numeric constant. The numeric constant must be enclosed in parentheses. The symbol is a microinstruction Field name used in the assembly. Sub-field names may not be used. The numeric constant specifies a bit to be selected from the field. The bits of a field are numbered right to left. Bit zero is the low order bit of the Field. Each Field operand selects a single bit from the microinstruction

The microinstruction bits selected by the Field operands are placed in the PROM module word in the order of the Field operand list. The bit from the first Field operand is placed in the leftmost (high order) bit position of the PROM module word. Normally the number of Field operands must equal the PROM module word width (defined by a previous MEMORY statement).

Unprogrammed bits may be assigned constant values with the SELECT statements. This is accomplished by using a numeric constant in place of a Field operand to set the value of a bit in the PROM module word. The value of the numeric constant must be zero or one. A constant bit specification can be used in the same manner as a Field operand.

The SELECT statement may also specify that the bit value of a Field operand be inverted when it is placed in the object word. This is specified by preceding the Field operand with the keyword – INVERT. If a group of contiguous Field operands (perhaps all Field operands in the SELECT statement) are to be inverted, they are enclosed in parentheses following the INVERT keyword. All bits in the group are inverted including constant bit specifications and already inverted Field operands (or groups). Parentheses may also be used without the INVERT keyword to group Field operands for readability.

Examples: Inverted Fields.

```
SELECT OP(4) OP(3) OP(2) OP(1) OP(0)
   INVERT KB(0);

SELECT INVERT (R(2) R(1) R(0) 1
   "UNPROGRAMMED");
```

When the extended feature of multiple microinstruction formats is used, a SELECT statement must be included for each microinstruction format which occurs in the object module. All field names in a SELECT statement must be from the same microinstruction format.

When the microinstruction format occupies more than one word in an assembly memory block, all words of the multiple word format must be specified in the SELECT statement. In this case, the number of Field operands is a multiple of the PROM module word width. Multiple words are specified in a left to right manner with the Field operands for the first word specified first followed immediately by the Field operands for the second word and so forth.

In the case of a look-up table PROM or a Field expansion PROM, where the Program Section contains data for only one PROM, the SELECT Statement must request all of the bits in the Program Section.

Occasionally, the microprogram will contain fewer instructions than the length of the PROM module. Since the Micro Format Program must output data to every PROM address, a default value for non-programmed locations may be defined. This is done by including an additional SELECT statement with the desired default value as the statement's only operand. If an additional default SELECT statement is not included, un-programmed locations will be filled with zeros.

Example of Default SELECT Statement:

```
SELECT          "DEFAULT ALL ONES LEAVES
(11111111)       PROM UN-PROGRAMMED" ;
```

As an example of assigning a microprogram to firmware, consider a microprogram that consists of 512 microinstructions, each 24 bits wide. A logical choice of Control Store PROMs would be three 82S115s which are 512 x 8 bipolar PROMs. In this case, a nine-bit address shared by the three PROMs would produce 512 24-bit microinstruction words.

Figure 5.4 illustrates a possible solution to the above problem of assigning a 512 x 24-bit microprogram to Control Store PROMs. The control Field FUNCTION has been vertically divided into two 256 x 8 PROM modules. A section of the command file assigning PROM modules one and two are presented in the figure.

MICROINSTRUCTION

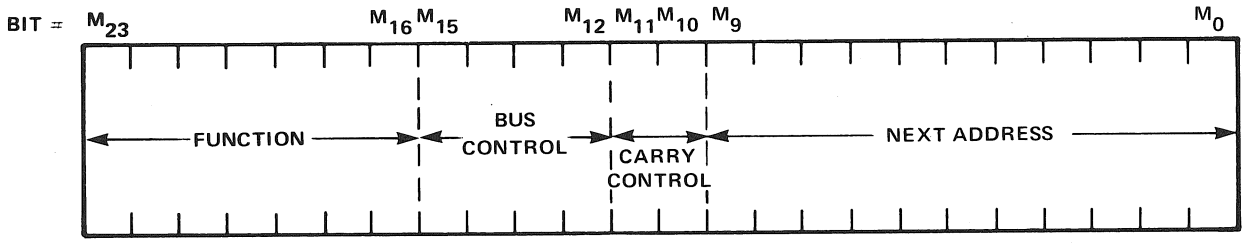


FIGURE 5.4(A)

PROM MODULES

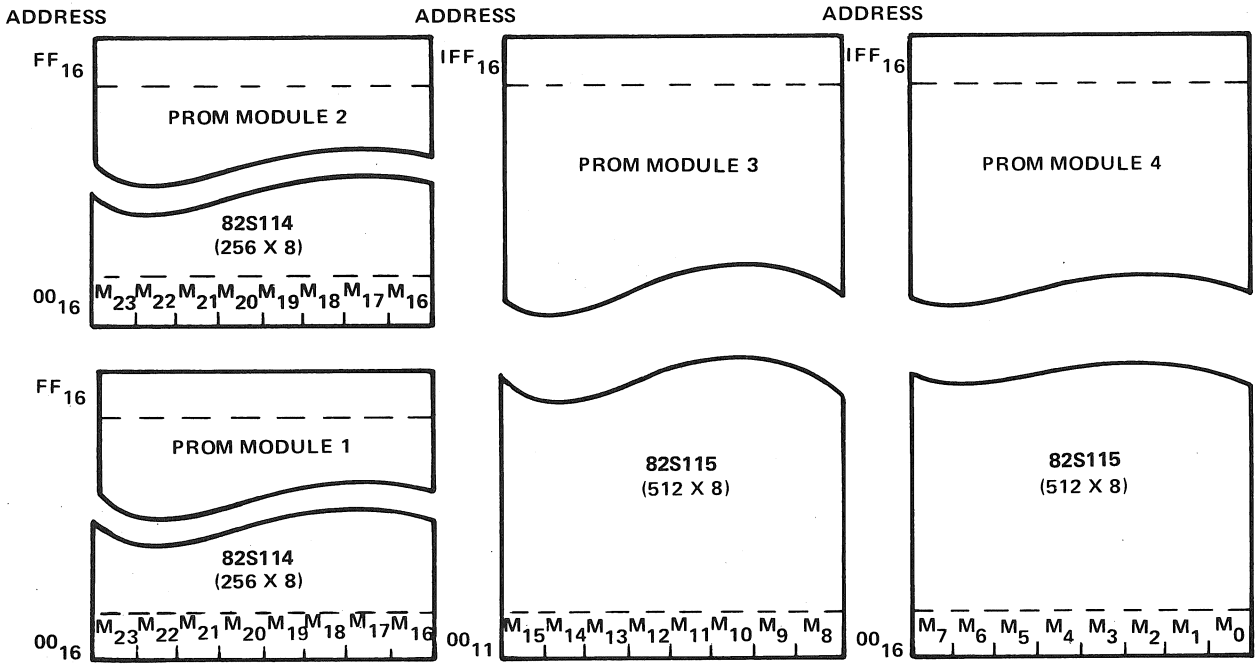


FIGURE 5.4 (B)

```
MEMORY   WIDTH 8   LENGTH 256     "PROM MODULE ONE" ;

OUTPUT   FROM 0   TO 255          OF MICROPROGRAM ;

         SELECT  FUNCTION(7)  FUNCTION(6)  FUNCTION(5)  FUNCTION(4)
                 FUNCTION(3)  FUNCTION(2)  FUNCTION(1)  FUNCTION(0) ;

MEMORY   WIDTH 8   LENGTH 256     "PROM MODULE TWO" ;

OUTPUT   FROM 256 TO 511          OF MICROPROGRAM ;

         SELECT  FUNCTION(7)  FUNCTION(6)  FUNCTION(5)  FUNCTION(4)
                 FUNCTION(3)  FUNCTION(2)  FUNCTION(1)  FUNCTION(0) ;
```

FIGURE 5.4 (C)

FIGURE 5.4

FORMATTING THE PAPER TAPE OUTPUT

When the microprogram and any auxiliary PROM programs have been properly assigned to PROM modules, the Micro Format Program needs only to know how the object data should be formatted to punch the PROM programming tapes. The PROM modules are formatted with two statements: the FORMAT Statement and the INSERT Statement. The FORMAT Statement specifies how each word of a PROM module is formatted on the output tape. The INSERT Statement allows the microprogrammer to punch tape beginning and ending control characters as well as intersperse comments within the tape output (the latter only if comment characters won't confuse the PROM programmer).

The FORMAT Statement

The FORMAT Statement specifies the format of the PROM module words (either binary or hexadecimal) and specifies optional framing characters if they are required. The format of the FORMAT Statement is presented in Figure 5.5.

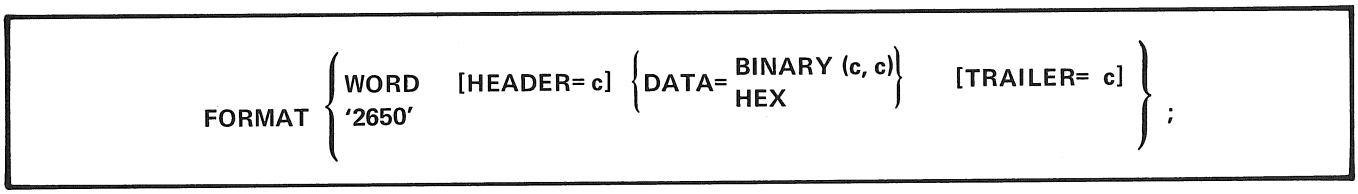


FIGURE 5.5

Output formats are selected with the keyword – WORD. In word format, no addresses are used. Intermediate object text is formatted into PROM module words (word width is defined by the MEMORY statement). The PROM module words are output starting at word address zero. Each data word may be formatted in hexadecimal or binary and may have framing characters. Additionally, in binary, arbitrary characters can be chosen to represent zero and one. Using the word format options, several common object output formats can be built, including SMS format and BNPF code.

Following the WORD keyword is a specification of the output format for memory words. This specification must be enclosed in parentheses. Within the parentheses is a sequence of one to three operands. Each operand consists of a keyword followed by an equal (=) followed by a value. The operands are as follows:

HEADER – This operand specifies the beginning frame characters on each word.

DATA – This operand specifies the output format of the object word.

TRAILER – This operand specifies the ending frame characters for each word.

The HEADER operand consists of HEADER= followed by a quoted string. The characters in the string will be placed in the front of each output word. If present, the HEADER operand must be the first operand. If the HEADER operand is omitted, *no* characters will be placed in front.

The DATA operand consists of DATA= followed by a format description. It may not be omitted. The description may be the keyword – HEX alone, indicating hexadecimal format. In HEX format, 0 through 9, A through F represent 0 through 9, 10 through 15, respectively. The description may also be the keyword – BINARY. When BINARY format is selected, the output characters for zero and one must be specified. These are specified with two quoted strings separated by a comma. The quoted string list must follow the BINARY keyword and be enclosed in parentheses. The first string will be used for zero, the second for 1. Each string must specify a single character.

In BINARY format, the number of characters output for a word is equal to the memory word width. In HEX format, the number of hexadecimal digits output for a word is computed from the memory word length. For example, if the memory word width is 8, two hexadecimal digits are used; if the width is 4, one is used.

The TRAILER operand consists of TRAILER= followed by a quoted string. The characters in the string will be placed after each output word. If present, the TRAILER must be the last operand (follows the DATA operand). If omitted, no characters will be placed after the output word.

The output words are arranged on object output records according to the length of the output word format. This width is equal to the number of characters (if any) in the header and trailer plus the number of characters in the data format. Using this length and the object output file record length, the number of words placed on each record is

determined. The number of words per record is always a power of two (2, 4, 8, 16, etc.). For example, an 8-bit word in SMS format (see the examples below) would be formatted 16 words per record with a record size of 80. In BNPF format, it would be formatted 8 words per record. The remaining portion (if any) of each record is left blank. If blanks are desired between words, they should be included in the header or trailer strings.

It may happen that more than one output format are required. Since the Micro Format Program reads the command file line by line, a new format defined by a FORMAT statement begins when the FORMAT Statement is encountered. Any PROM modules defined after a FORMAT Statement follow the format it has defined.

The Micro Format Program has intrinsic mechanisms for generating 2650 Absolute Object Code which can be read by the Signetics TWIN (Test Ware Instrument – A microprocessor system development instrument) when it is used to program PROMs. The 2650 Absolute Object Code format is selected by following the Op-Code FORMAT with the character constant string, "2650." (See Appendix D for a full description of the 2650 Absolute Object Format.)

The 2650 Absolute Object Format may use an execution address. This address is passed to the Micro Format Program by the END Statement of the microprogram. In this case, the terminating END Statement of the Micro-program Section may have the execution address as an operand.

When the 2650 format is used, the length specification in the MEMORY Statement may be omitted. If the length specification is omitted the FROM address must equal zero. In this mode of operation, data will be placed in programmed addresses only.

Examples: FORMAT Statements.

```
FORMAT '2650' "2650 ABSOLUTE OBJECT
  FORMAT";

FORMAT WORD (DATA=HEX TRAILER= ' ' ' ')
  "SMS FORMAT";

FORMAT WORD (HEADER='B' DATA=BINARY
  ('N', 'P') TRAILER='F') "BNPF CODE";

FORMAT WORD (HEADER= 'B' DATA=BINARY
  ('N', 'P') TRAILER='F') "BNPF CODE WITH A
  BLANK BETWEEN WORDS";

FORMAT WORD (DATA=HEX) "UNFRAMED
  HEXADECIMAL OUTPUT";
```

The INSERT Statement

The INSERT Statement requests output of literal data onto the PROM programming tape. The format of the INSERT Statement is given in Figure 5.6.

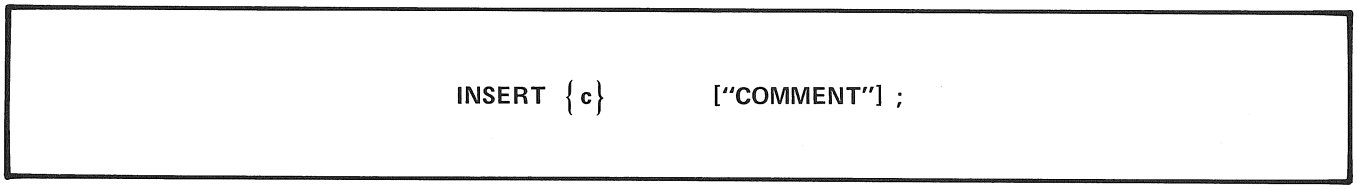


FIGURE 5.6

The statement consists of the INSERT opcode followed by a character constant. The characters of the constant are output as a record in the object output file. If the number of characters in the constant is not equal to the record length of the object output file, it is blank-filled or truncated, as necessary, on the right.

The INSERT Statement is used to generate literal object records. These records may come before and after PROM modules. They are used in the object output file to separate and identify PROM modules and to provide framing characters or records for PROM modules.

Examples:

```
INSERT 'EMULATOR MICRO PROGRAM'
  "OBJECT TITLE";

INSERT '*********' "OBJECT MODULE
  SEPARATOR";
```

The END Statement

The command file is terminated with the END Statement. The END Statement's format is illustrated in Figure 5.7.

The END Statement consists only of the END Op-Code.

```
END          ["COMMENT"] ;
```

FIGURE 5.7

THE MICRO FORMAT PROGRAM LISTINGS

The listing file for the Micro Format Program has the same structure as the listing file of the Micro Assembly Program. The first heading line has the following format:

```
SIGNETICS BIPOLAR MICRO FORMATTER(A)
                ********************** PAGE 00000
```

The row of asterisks contain the title of the listing section. The second heading describes the information in the listing and depends on the listing section.

The first section of the listing file is a formatted listing of the command file input. The listing records consist of a command file line number followed by the first 72 positions of a command file record. Error records may be interspersed in the first listing section. The first heading line contains – FORMAT CONTROL INPUT. The second heading line is:

```
LINE        IMAGE
```

The second section of the listing file is a summary listing. The summary listing has two sub-sections, each beginning on a new page. The first sub-section is a listing of the PROM modules produced. Each PROM module is given a number and the following information is listed – module number, module word width, module length in words, object format (2650, HEX or BINARY), beginning address in the Program Section, Program Section name. In this sub-section, the first heading line contains – OBJECT MODULE SUMMARY. The second heading line is:

```
MODULE WIDTH LENGTH FORMAT FROM PROGRAM
```

The second sub-section of the summary listing lists allocation of microinstruction fields to PROM modules. For each bit of each field the following information is listed – microinstruction format ID, field name and bit number of the field, object number and bit number in the object word. In this sub-section, the first heading line contains – MICROINSTRUCTION FIELD SUMMARY. The second heading is:

```
ID FIELD          OBJECT MODULE
```

APPENDIX A

SOURCE TOGGLES – SETTING PROGRAM PARAMETERS

The microprogrammer is given control of various program parameters with source toggles. These parameters include the Fortran device number of the Input/Output files, the number of lines in a page of output listing, the maximum word size (in bits) of numerical values and control of cross reference listing (whether or not it is produced).

Most toggles may be set anywhere in the source input to the two programs, even between lines of a single statement. The exceptions are toggles that must be set before any assembly language statement in the Micro Assembly Program and toggles that must be set before any command file statements in the Micro Format Program.

Source toggles are set with toggle operands. The format of the toggle operands is given in Figure A.1.

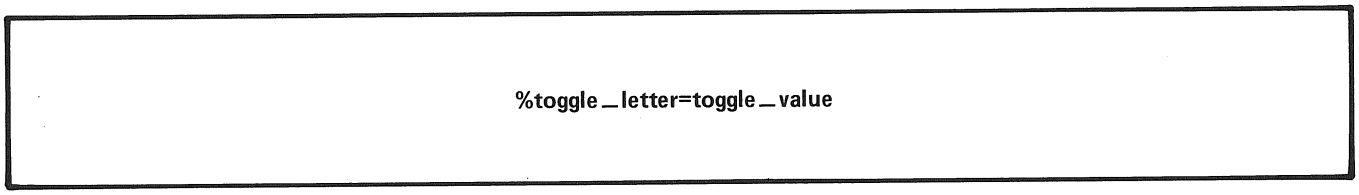


FIGURE A.1

The first character of a source line containing a toggle operand is a percent sign (%). The percent sign is followed by a single letter which identifies the toggle being set. The toggle letter is followed by an equal sign (=) delimiter and the value that the toggle is to be set to. There may be no embedded blanks in the toggle operand string. The toggle value is set to a decimal numeric constant.

Multiple toggles may be set on a single line of source, as each percent sign acts as a delimiter between adjacent toggle operands.

If a toggle operand that must set before any source statement is set after the first language element of the first statement of the source, it is ignored and no error indication is given. Other illegal toggle operands are also ignored with no error indication.

Toggles are not reproduced on the source listings.

MICRO ASSEMBLER TOGGLES

The following Micro Assembler toggles may be set anywhere in the source file:

Toggle Name	Description
P	Size in lines of a page in the listing file. Default value is 60. The minimum value is 10.
X	If the toggle is set to a non-zero value, the symbol cross reference listing will be generated for all of the source input following the toggle. If the value is zero, no cross reference listing will be produced. Default value is 1. Thus, the X toggle permits selective generation of the cross reference listing.
S	Fortran device number for source file. Default value is 1.
O	Fortran device number for object file. Default value is 2.
L	Fortran device number for listing file. Default value is 4.
I	Fortran device number for instrinsic file. See Appendix C for a description of the intrinsic file. Default value is 3.

The following Micro Assembler toggles must be set before any assembly language statement:

Toggle Name	Description
A,B	Fortran device numbers for two assembler work files. See Appendix C. Default values are 7 for A and 8 for B.
M	Maximum number of bits in a memory word or instruction format in the current assembly. The default value is 128. The maximum value is 4095 and the minimum value is 32.
F	Number of bits in the Fortran integer word on the source machine. This will optimize use of internal space within the micro assembly program. The default value is 16. The minimum value is 13. The Micro Assembler assumes the maximum positive integer value to be $2^{(\%F-1)}-1$.

Examples: Micro Assembly toggle records (values are default values).

%P=60 %X=1 %S=1 %O=2 %L=4 %I=3

%A=7 %B=8 %M=128 %F=16

MICRO FORMAT TOGGLES

The following Micro Format toggles may be set anywhere in the format control file:

Toggle Name	Description
P	Size in lines of a page in the listing file. Default value is 60. The minimum value is 10.
C	Fortran device number for the format control file. Default value is 1.
O	Fortran device number for the loadable object output file. Default value is 2.
I	Fortran device number for the intermediate object input file. Default value is 3.
L	Fortran device number for the listing file. Default value is 4.

The following Micro Format toggles must be set before any command file statements:

Toggle Name	Description
M	Maximum number of bits in any object word or micro instruction in the intermediate object file. The default value is 128. The maximum value is 4095 and the minimum value is 32.
F	Number of bits in the Fortran integer word on the format machine. Default value is 16. The minimum value is 13.

Examples: Micro Format toggle records (values are default values).

%P=60 %C=1 %O=2 %I=3 %L=4

%M=128 %F=16

APPENDIX B – INTRINSIC MICROP EQUIVALENT SOURCE INPUT

The following three figures present source input that if included in the user's source, would produce results equivalent to calling the intrinsic microps with INTRINSIC Statements.

```
SIGNETICS BIPOLAR MICRO ASSEMBLER(A)   8X02 INTRINSIC MICROPS.        PAGE 00002
  ADDR: ACF      BA

00007         "8X02 INTRINSIC MICROPS"
00008    MICROP TSK ASSIGN ACF=000B;
00009    MICROP INC ASSIGN ACF=001B;
00010    MICROP BLT ASSIGN ACF=010B;
00011    MICROP POP ASSIGN ACF=011B;
00012    MICROP BSR ASSIGN ACF=100B;
00013    MICROP PLP ASSIGN ACF=101B;
00014    MICROP BRT ASSIGN ACF=110B;
00015    MICROP RST ASSIGN ACF=111B;
```

FIGURE B.1

```
SIGNETICS BIPOLAR MICRO ASSEMBLER(A)   3002 INTRINSIC MICROPS.        PAGE 00002
  ADDR:     F      KB

00007         "3002 INTRINSIC EQU'S"
00008  R0   :      EQU   00; "3002 REGISTER 00"
00009  R1   :      EQU   01; "3002 REGISTER 01"
00010  R2   :      EQU   02; "3002 REGISTER 02"
00011  R3   :      EQU   03; "3002 REGISTER 03"
00012  R4   :      EQU   04; "3002 REGISTER 04"
00013  R5   :      EQU   05; "3002 REGISTER 05"
00014  R6   :      EQU   06; "3002 REGISTER 06"
00015  R7   :      EQU   07; "3002 REGISTER 07"
00016  R8   :      EQU   08; "3002 REGISTER 08"
00017  R9   :      EQU   09; "3002 REGISTER 09"
00018  T    :      EQU   10; "3002 REGISTER T"
00019  AC   :      EQU   11; "3002 REGISTER AC"
00020         "3002 INTRINSIC MICROPS"
00021         "R GROUP I"
00022    MICROP ARAMAC(@R) ASSIGN IF @R EQ AC OR @R EQ T THEN F=02H+@R ELSE F=00H+@R FI;
00023    MICROP ILR(@R) ASSIGN ARAMAC(@R) DEFAULT KB=0;
00024    MICROP ALR(@R) ASSIGN ARAMAC(@R) DEFAULT KB=-1;
00025    MICROP MARK(@R)   ASSIGN IF @R EQ AC OR @R EQ T THEN F=12H+@R ELSE F=10H+@R FI;
00026    MICROP MINR(@R) ASSIGN MARK(@R) DEFAULT KB=0;
00027    MICROP MARS(@R) ASSIGN MARK(@R) DEFAULT KB=-1;
00028    MICROP ASAM(@R)   ASSIGN IF @R EQ AC OR @R EQ T THEN F=22H+@R ELSE F=20H+@R FI;
00029    MICROP CSR(@R) ASSIGN ASAM(@R) DEFAULT KB=0;
00030    MICROP ASA(@R) ASSIGN ASAM(@R) DEFAULT KB=-1;
00031    MICROP ARAM(@R)   ASSIGN IF @R EQ AC OR @R EQ T THEN F=32H+@R ELSE F=30H+@R FI;
00032    MICROP INR(@R) ASSIGN ARAM(@R) DEFAULT KB=0;
00033    MICROP ARA(@R) ASSIGN ARAM(@R) DEFAULT KB=-1;
00034    MICROP NRAM(@R)   ASSIGN IF @R EQ AC OR @R EQ T THEN F=42H+@R ELSE F=00H+@R FI;
00035    MICROP CLR(@R) ASSIGN NRAM(@R) DEFAULT KB=0;
00036    MICROP NRA(@R) ASSIGN NRAM(@R) DEFAULT KB=-1;
00037    MICROP LTRM(@R)   ASSIGN IF @R EQ AC OR @R EQ T THEN F=52H+@R ELSE F=50H+@R FI;
00038    MICROP LTR(@R) ASSIGN LTRM(@R) DEFAULT KB=-1;
```

FIGURE B.2

```
00039    MICROP ORAM(@R)    ASSIGN IF @R EQ AC OR @R EQ T THEN F=62H+@R ELSE F=60H+@R FI;
00040    MICROP NOP(@R) ASSIGN ORAM(@R) DEFAULT KB=0;
00041    MICROP ORA(@R) ASSIGN ORAM(@R) DEFAULT KB=-1;
00042    MICROP XRAM(@R)    ASSIGN IF @R EQ AC OR @R EQ T THEN F=72H+@R ELSE F=70H+@R FI;
00043    MICROP LCR(@R) ASSIGN XRAM(@R) DEFAULT KB=0;
00044    MICROP XRA(@R) ASSIGN XRAM(@R) DEFAULT KB=-1;
00045        "R GROUP II"
00046    MICROP AMAM(@R)    ASSIGN IF @R EQ AC OR @R EQ T THEN F=00H+@R FI;
00047    MICROP AM(@R)  ASSIGN AMAM(@R) DEFAULT KB=0;
00048    MICROP AMA(@R) ASSIGN AMAM(@R) DEFAULT KB=-1;
00049    MICROP MAMK(@R)    ASSIGN IF @R EQ AC OR @R EQ T THEN F=10H+@R FI;
00050    MICROP MAM(@R) ASSIGN MAMK(@R) DEFAULT KB=0;
00051    MICROP MAMS(@R) ASSIGN MAMK(@R) DEFAULT KB=-1;
00052    MICROP ASAMX(@R)   ASSIGN IF @R EQ AC OR @R EQ T THEN F=20H+@R FI;
00053    MICROP AMAMX(@R)   ASSIGN IF @R EQ AC OR @R EQ T THEN F=30H+@R FI;
00054    MICROP NMAM(@R)    ASSIGN IF @R EQ AC OR @R EQ T THEN F=40H+@R FI;
00055    MICROP NMA(@R) ASSIGN NMAM(@R) DEFAULT KB=-1;
00056    MICROP LTMM(@R)    ASSIGN IF @R EQ AC OR @R EQ T THEN F=50H+@R FI;
00057    MICROP LTM(@R) ASSIGN LTMM(@R) DEFAULT KB=-1;
00058    MICROP OMAM(@R)    ASSIGN IF @R EQ AC OR @R EQ T THEN F=60H+@R FI;
00059    MICROP LM(@R)  ASSIGN OMAM(@R) DEFAULT KB=0;
00060    MICROP OMA(@R) ASSIGN OMAM(@R) DEFAULT KB=-1;
00061    MICROP XMAM(@R)    ASSIGN IF @R EQ AC OR @R EQ T THEN F=70H+@R FI;
00062    MICROP LCM(@R) ASSIGN XMAM(@R) DEFAULT KB=0;
00063    MICROP XMA(@R) ASSIGN XMAM(@R) DEFAULT KB=-1;

SIGNETICS BIPOLAR MICRO ASSEMBLER(A)    3002 INTRINSIC MICROPS.        PAGE 00003
  ADDR:     F      KB

00064        "R GROUP III"
00065    MICROP SRA(@R) ASSIGN IF @R EQ AC OR @R EQ T THEN F=04H+@R FI DEFAULT KB=0;
00066    MICROP ACAKAM(@R) ASSIGN IF @R EQ AC OR @R EQ T THEN F=14H+@R FI;
00067    MICROP ACA(@R) ASSIGN ACAKAM(@R) DEFAULT KB=0;
00068    MICROP AAS(@R) ASSIGN ACAKAM(@R) DEFAULT KB=-1;
00069    MICROP ASIM(@R)    ASSIGN IF @R EQ AC OR @R EQ T THEN F=24H+@R FI;
00070    MICROP ASI(@R) ASSIGN ASIM(@R) DEFAULT KB=-1;
00071    MICROP AAIM(@R)    ASSIGN IF @R EQ AC OR @R EQ T THEN F=34H+@R FI;
00072    MICROP AAI(@R) ASSIGN AAIM(@R) DEFAULT KB=-1;
00073    MICROP NAIM(@R)    ASSIGN IF @R EQ AC OR @R EQ T THEN F=44H+@R FI;
00074    MICROP NAI(@R) ASSIGN NAIM(@R) DEFAULT KB=-1;
00075    MICROP LTAM(@R)    ASSIGN IF @R EQ AC OR @R EQ T THEN F=54H+@R FI;
00076    MICROP OAIM(@R)    ASSIGN IF @R EQ AC OR @R EQ T THEN F=64H+@R FI;
00077    MICROP OAI(@R) ASSIGN OAIM(@R) DEFAULT KB=-1;
00078    MICROP XAIM(@R)    ASSIGN IF @R EQ AC OR @R EQ T THEN F=74H+@R FI;
00079    MICROP XAI(@R) ASSIGN XAIM(@R) DEFAULT KB=-1;
```

FIGURE B.2 (continued)

```
SIGNETICS BIPOLAR MICRO ASSEMBLER(A)    2901 INTRINSIC MICROPS.        PAGE 00002
  ADDR:    A     B  DST FUN SRC

00010        "2901 INTRINSIC EQU'S"
00011  R0  :      EQU  00; "2901 REGISTER 00"
00012  R1  :      EQU  01; "2901 REGISTER 01"
00013  R2  :      EQU  02; "2901 REGISTER 02"
00014  R3  :      EQU  03; "2901 REGISTER 03"
00015  R4  :      EQU  04; "2901 REGISTER 04"
00016  R5  :      EQU  05; "2901 REGISTER 05"
00017  R6  :      EQU  06; "2901 REGISTER 06"
00018  R7  :      EQU  07; "2901 REGISTER 07"
00019  R8  :      EQU  08; "2901 REGISTER 08"
00020  R9  :      EQU  09; "2901 REGISTER 09"
00021  R10 :      EQU  10; "2901 REGISTER 10"
```

FIGURE B.3

```
00022  R11 :      EQU  11; "2901 REGISTER 11"
00023  R12 :      EQU  12; "2901 REGISTER 12"
00024  R13 :      EQU  13; "2901 REGISTER 13"
00025  R14 :      EQU  14; "2901 REGISTER 14"
00026  R15 :      EQU  15; "2901 REGISTER 15"
00027  Q   :      EQU -16; "2901 REGISTER Q"
00028  STR :      EQU   3; "2901 DESTINATION : STORE RAM"
00029  STQ :      EQU   0; "2901 DESTINATION : STORE Q"
00030  SLR :      EQU   5; "2901 DESTINATION : SHIFT LEFT RAM"
00031  SLRQ:      EQU   4; "2901 DESTINATION : SHIFT LEFT RAM + Q"
00032  SRR :      EQU   7; "2901 DESTINATION : SHIFT RIGHT RAM"
00033  SRRQ:      EQU   6; "2901 DESTINATION : SHIFT RIGHT RAM + Q"
00034  Y   :      EQU   1; "2901 DESTINATION : STORE RAM + OUTPUT A REGISTER"
00035       "2901 INTRINSIC MICROPS"
00036       "2 ADDRESS INSTRUCTIONS"
00037   MICROP a2R(aR,aS,aM,aD,aY)  ASSIGN A=aR IF aS NE Q
00038     THEN SRC=1 B=aS ELSE SRC=0 B=aD FI IF aY EQ 16 THEN DST=aM ELSE DST=2 FI;
00039   MICROP  AR(aR,aS) aM=1(aD=aS),aY=16(aA)  ASSIGN FUN=0 a2R(aR,aS,aM,aD,aY);
00040   MICROP RSR(aR,aS) aM=1(aD=aS),aY=16(aA)  ASSIGN FUN=1 a2R(aR,aS,aM,aD,aY);
00041   MICROP  SR(aR,aS) aM=1(aD=aS),aY=16(aA)  ASSIGN FUN=2 a2R(aR,aS,aM,aD,aY);
00042   MICROP ORR(aR,aS) aM=1(aD=aS),aY=16(aA)  ASSIGN FUN=3 a2R(aR,aS,aM,aD,aY);
00043   MICROP  NR(aR,aS) aM=1(aD=aS),aY=16(aA)  ASSIGN FUN=4 a2R(aR,aS,aM,aD,aY);
00044   MICROP NCR(aR,aS) aM=1(aD=aS),aY=16(aA)  ASSIGN FUN=5 a2R(aR,aS,aM,aB,aY);
00045   MICROP  XR(aR,aS) aM=1(aD=aS),aY=16(aA)  ASSIGN FUN=6 a2R(aR,aS,aM,aD,aY);
00046   MICROP XCR(aR,aS) aM=1(aD=aS),aY=16(aA)  ASSIGN FUN=7 a2R(aR,aS,aM,aD,aY);
00047       "1 ADDRESS INSTRUCTIONS"
00048   MICROP  a1R(aS,aM,aD,aY,aA)  ASSIGN B=aD IF aS NE Q
00049     THEN IF aS EQ aD THEN SRC=3 A=aA ELSE SRC=4 A=aS FI ELSE SRC=2 A=aA FI
00050     IF aY EQ 16 THEN DST=aM ELSE DST=2 FI;
00051   MICROP  IR(aS) aM=1(aD=aS),aY=16(aA=aS)  ASSIGN FUN=0 a1R(aS,aM,aD,aY,aA);
00052   MICROP  DR(aS) aM=1(aD=aS),aY=16(aA=aS)  ASSIGN FUN=1 a1R(aS,aM,aD,aY,aA);
00053   MICROP LNR(aS) aM=1(aD=aS),aY=16(aA=aS)  ASSIGN FUN=2 a1R(aS,aM,aD,aY,aA);
00054   MICROP  LR(aS) aM=1(aD=aS),aY=16(aA=aS)  ASSIGN FUN=3 a1R(aS,aM,aD,aY,aA);
00055   MICROP LCR(aS) aM=1(aD=aS),aY=16(aA=aS)  ASSIGN FUN=7 a1R(aS,aM,aD,aY,aA);
00056       "1 ADDRESS INSTRUCTIONS"
00057   MICROP a1D(aR,aM,aD,aY,aA)  ASSIGN B=aD IF aR NE Q
00058     THEN SRC=5 A=aR ELSE SRC=6 A=aA FI IF aY EQ 16 THEN DST=aM ELSE DST=2 FI;
00059   MICROP  AD(aR) aM=1(aD=aR),aY=16(aA=aR)  ASSIGN FUN=0 a1D(aR,aM,aD,aY,aA);
00060   MICROP RSD(aR) aM=1(aD=aR),aY=16(aA=aR)  ASSIGN FUN=1 a1D(aR,aM,aD,aY,aA);
00061   MICROP  SD(aR) aM=1(aD=aR),aY=16(aA=aR)  ASSIGN FUN=2 a1D(aR,aM,aD,aY,aA);
00062   MICROP  OD(aR) aM=1(aD=aR),aY=16(aA=aR)  ASSIGN FUN=3 a1D(aR,aM,aD,aY,aA);
00063   MICROP  ND(aR) aM=1(aD=aR),aY=16(aA=aR)  ASSIGN FUN=4 a1D(aR,aM,aD,aY,aA);
00064   MICROP NCD(aR) aM=1(aD=aR),aY=16(aA=aR)  ASSIGN FUN=5 a1D(aR,aM,aD,aY,aA);
00065   MICROP  XD(aR) aM=1(aD=aR),aY=16(aA=aR)  ASSIGN FUN=6 a1D(aR,aM,aD,aY,aA);
00066   MICROP XCD(aR) aM=1(aD=aR),aY=16(aA=aR)  ASSIGN FUN=7 a1D(aR,aM,aD,aY,aA);

SIGNETICS BIPOLAR MICRO ASSEMBLER(A)   2901 INTRINSIC MICROPS.        PAGE 00003
  ADDR:    A     B  DST FUN SRC

00067       "0 ADDRESS INSTRUCTIONS"
00068   MICROP a0D(aM,aD,aY,aA)  ASSIGN SRC=7 B=aD
00069     IF aY EQ 16 THEN DST=aM A=0 ELSE DST=2 A=aA FI;
00070   MICROP  ID aM=1(aD=0),aY=16(aA=0)  ASSIGN FUN=0 a0D(aM,aD,aY,aA);
00071   MICROP LND aM=1(aD=0),aY=16(aA=0)  ASSIGN FUN=1 a0D(aM,aD,aY,aA);
00072   MICROP  DD aM=1(aD=0),aY=16(aA=0)  ASSIGN FUN=2 a0D(aM,aD,aY,aA);
00073   MICROP  LD aM=1(aD=0),aY=16(aA=0)  ASSIGN FUN=3 a0D(aM,aD,aY,aA);
00074   MICROP  LZ aM=1(aD=0),aY=16(aA=0)  ASSIGN FUN=4 a0D(aM,aD,aY,aA);
00075   MICROP LCD aM=1(aD=0),aY=16(aA=0)  ASSIGN FUN=7 a0D(aM,aD,aY,aA);
```

FIGURE B.3 (continued)

APPENDIX C – INSTALLATION CONSIDERATIONS

COMPILATION

The Fortran source deck for the Micro Assembly and Micro Format Programs can be compiled with any standard ANSI Fortran compiler. Prior to compilation, the defaults for source toggles can be altered in the Fortran source. The record lengths for each file may also be altered. This change is described in the Installation Manual provided with Micro Assembler Source decks (or tapes). The toggles and their defaults are described in Appendix A.

EXECUTION – MICRO ASSEMBLY PROGRAM

During execution, the Micro Assembly Program utilizes several I/O files. All files are sequential and consist of 80 character fixed length records. Except for the work files, each file is accessed as an input file or output file only. All input files may be rewound and reread as required.

Source File

This file is an input only file which contains the source input for the Micro Assembly Program. The toggle for this file is %S.

Object File

This file is an output only file which contains the intermediate object output of the Micro Assembly Program. The toggle for this file is %O.

Listing File

This file is an output only file which contains the listing output of the assembly. The toggle for this file is %L.

Work Files

There are two work files which are used to sort the symbol cross reference information. Both files are used for input and output. The toggles for these files are %A and %B.

Intrinsic File

The intrinsic file is an input only file which contains the intrinsic microp definitions. The toggle for this file is %I. This file contains the intrinsic microps for the 8X02, N3002 and the N2901-1. The intrinsic deck is available with the Micro Assembler source.

Each intrinsic definition block consists of a header record followed by statements that define the intrinsic microps and value symbol assignments. The header record begins with a percent sign (%). The percent sign is followed by a self-defining-constant which identifies the intrinsic definition block. The remainder of the header is available for user comments. Each intrinsic definition block is terminated by the header record of the next intrinsic definition block. The last intrinsic definition block is terminated by a record consisting of a single percent sign followed by blanks. Intrinsic files may not contain toggle records.

EXECUTION – MICRO FORMAT PROGRAM

During execution, the Micro Format Program utilizes several I/O files. All files are sequential and consist of 80 character fixed length records. Each file is accessed as either an input only file or an output only file. The intermediate object file (input) may be rewound and reread as required.

Command File

This file is an input only file which contains the command file for the Micro Format Program. The toggle for this file is %C.

Intermediate Object File

This file is an input only file which contains the intermediate object file output from the Micro Assembly Program. The toggle for this file is %I.

Listing File

This file is an output only file which contains the listing output of the Micro Format Program. The toggle for this file is %L.

Object Output File

This is an output only file that contains the object output (PROM programming tapes) from the Micro Format Program. The toggle for this file is %O.

APPENDIX D – RESERVED WORDS OF THE MICRO ASSEMBLER LANGUAGE

*MICRO ASSEMBLY PROGRAM RESERVED WORDS

Statement Op-Codes:

INSTRUCTION	MICROP	DCL
END	INTRINSIC	EQU
FIELD	PROGRAM	SET
ORG	OBJECT	LIST
TITLE	SPACE	EJECT
FORMAT		

Keywords:

LENGTH	DEFAULT	ASSIGN
WIDTH	ON	OFF
SOURCE	IF	THEN
ELSE	FI	

Expression Operators:

NOT	AND	OR
XOR	SHL	SHR
EQ	NE	GT
LT	GE	LE

*MICRO FORMAT PROGRAM RESERVED WORDS

Statement Op-Codes:

FORMAT	MEMORY	OUTPUT
SELECT	INSERT	END

Keywords:

WORD	HEADER	DATA
TRAILER	HEX	BINARY
WIDTH	LENGTH	FROM
TO	OF	INVERT

APPENDIX E – 2650 ABSOLUTE OBJECT FORMAT

(From Applications Memo SS51)

INTRODUCTION

The format for absolute code produced for the 2650 is described in this application note. The absolute object code is formatted into blocks. The first character of every block is a colon. Inside of a block, all the characters are hexadecimal, i.e., 0 to 9 or A to F, inclusive. Only non-printing ASCII control characters may occur within an interblock gap. These are the characters in the first two columns (columns 0 and 1) of the ASCII standard code table. A CR/LF is used within the interblock gap to reset the TTY or terminal after each block.

Each block is independent. For example, paper tape can be positioned prior to any block and a load started. The loading of absolute object code will be halted by:

A BCC error on the address + count fields
A BCC error on the data field
An incorrect block length
A non-hex character within the block

The block length field contains the number of bytes of actual data which is half the number of hex characters in the data field. While the size of the data field can range from 2 to 510 characters, a standard size of 60 characters has been established so that the tape may be easily generated and read on a variety of terminals and systems. A block length of zero indicates an End of File (EOF) block. The address field of an EOF block contains the start address of the loaded program.

The Block Control Character is 8 bits formed from the actual bytes and not from the ASCII characters. The bytes are in turn exclusive or'ed to the BCC byte, and then the BCC byte is left rotated one bit. It appears as two hex characters. Both the address and count fields and the data field are followed by a BCC character pair. The BCC prevents storing data at an invalid memory address or storing bad data into memory.

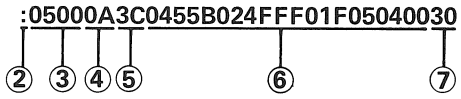
EXAMPLE: An object tape that loads ten bytes starting at location 500

```
:05000A3C0455B024FFF01F05040030
:000000
```

FORMAT

1. Interblock gap of any non-printing characters including spaces
2. Start of block character;
   a colon
3. Address field;
   four hex characters
4. Count field;
   two hex characters in range 0 to 1E
5. BCC for address and count fields;
   two hex characters
6. Data field;
   twice the value in the count field which is the number of memory locations loaded by the current block
7. BCC for the data field;
   two hex characters

EXAMPLE OF OBJECT FORMAT



2 – Start of block character (colon)
3 – Starting address for block (H'0500')
4 – Number of bytes in block (H'0A' = 10)
5 – BCC byte for fields 3 and 4 (H'3C')
6 – Data, two characters per byte
7 – BCC byte for field 6 (H'30')

APPENDIX F – ASCII CHARACTER SET

ACSII CHARACTER SET (7-BIT CODE)									
L.S. CHAR \ M.S. CHAR		0 000	1 001	2 010	3 011	4 100	5 101	6 110	7 111
0	0000	NUL	DLE	SP	0	@	P	`	p
1	0001	SOH	DC1	!	1	A	Q	a	q
2	0010	STX	DC2	"	2	B	R	b	r
3	0011	ETX	DC3	#	3	C	S	c	s
4	0100	EOT	DC4	$	4	D	T	d	t
5	0101	ENQ	NAK	%	5	E	U	e	u
6	0110	ACK	SYN	&	6	F	V	f	v
7	0111	BEL	ETB	'	7	G	W	g	w
8	1000	BS	CAN	(	8	H	X	h	x
9	1001	HT	EM	)	9	I	Y	i	y
A	1010	LF	SUB	*	:	J	Z	j	z
B	1011	VT	ESC	+	;	K	[	k	{
C	1100	FF	FS	,	<	L	\	l	¦
D	1101	CR	GS	–	=	M	]	m	}
E	1110	SO	RS	•	>	N	^	n	~
F	1111	SI	US	/	?	O	_	o	DEL

APPENDIX G – POWERS OF TWO TABLE

2^n	n	2^{-n}
1	0	1.0
2	1	0.5
4	2	0.25
8	3	0.125
16	4	0.062 5
32	5	0.031 25
64	6	0.015 625
128	7	0.007 812 5
256	8	0.003 906 25
512	9	0.001 953 125
1 024	10	0.000 976 562 5
2 048	11	0.000 488 281 25
4 096	12	0.000 244 140 625
8 192	13	0.000 122 070 312 5
16 384	14	0.000 061 035 156 25
32 768	15	0.000 030 517 578 125
65 536	16	0.000 015 258 789 062 5
131 072	17	0.000 007 629 394 531 25
262 144	18	0.000 003 814 697 265 625
524 288	19	0.000 001 907 348 632 812 5
1 048 576	20	0.000 000 953 674 316 406 25
2 097 152	21	0.000 000 476 837 158 203 125
4 194 304	22	0.000 000 238 418 579 101 562 5
8 388 608	23	0.000 000 119 209 289 550 781 25
16 777 216	24	0.000 000 059 604 644 775 390 625
33 554 432	25	0.000 000 029 802 322 387 695 312 5
67 108 864	26	0.000 000 014 901 161 193 847 656 25
134 217 728	27	0.000 000 007 450 580 596 923 828 125
268 435 456	28	0.000 000 003 725 290 298 461 914 062 5
536 870 912	29	0.000 000 001 862 645 149 230 957 031 45
1 073 741 824	30	0.000 000 000 931 322 574 615 478 515 625
2 147 483 648	31	0.000 000 000 465 661 287 307 739 257 812 5
4 294 967 296	32	0.000 000 000 232 830 643 653 869 628 906 25
8 589 934 592	33	0.000 000 000 116 415 321 826 934 814 453 125
17 179 869 184	34	0.000 000 000 058 207 660 913 467 407 226 562 5
34 359 738 368	35	0.000 000 000 029 103 830 456 733 703 613 281 25
68 719 476 736	36	0.000 000 000 014 551 915 228 366 851 806 640 625
137 438 953 472	37	0.000 000 000 007 275 957 614 183 425 903 320 312 5
274 877 906 944	38	0.000 000 000 003 637 978 807 091 712 951 660 156 25
549 755 813 888	39	0.000 000 000 001 818 989 403 545 856 475 830 078 125
1 099 511 627 776	40	0.000 000 000 000 909 494 701 772 928 237 915 039 062 5

APPENDIX H– MICRO ASSEMBLY PROGRAM SUMMARY

This appendix presents a condensed description of the Micro Assembly Program. It is included for quick reference once the microprogrammer is familiar with the basic structure of the Micro Assembly language.

Statements consist of:

- one or more labels, each followed by a colon (labels are optional)
- an Op-Code that identifies the type of statement
- one or more operands
- a statement terminator, the semicolon

The following is a generalized format for statements:

[label:] . . . { Op-Code } { operand } ;

Operands must be separated from the Op-Code and from each other by blanks.

The following symbols are used to describe the language:

[ ]	Brackets enclose language elements that are optional.
{ }	Braces enclose language elements that are not optional.
. . .	Ellipses (three periods) indicate that the preceding language element may be repeated one or more times.
____	A language element that is underlined must have been previously defined.
Option A Option B Option C	Two or more stacked elements indicate a choice. Any one of the stacked elements may be chosen.
UPPER CASE	Capitalized words are reserved words and must appear in the statement exactly as shown
lower case	Lower case words represent language elements that may be specified with expressions. All symbols used in expressions must be defined in the label field, (e.g., with an EQU statement). Lower case words are also used to specify a symbol used in the label field.
UPPER and lower case	Words that begin with capital letter specify symbols that are not defined in a label field (e.g., a field name).

The following special characters are used in the source code:

:	Indicates that the preceding symbol is a label or an absolute address.
;	Terminates a statement.
=	Used in Microinstruction Statements to: • assign a value to a field • pass a parameter to a MICROP • set a default value for a MICROP parameter
( )	Parenthesis are used for grouping language elements.

THE BASIC LANGUAGE

I. DEFINITION SECTION

{INSTRUCTION} {WIDTH numberofbits} ;

{FIELD Name} {WIDTH numberofbits}
[DEFAULT value] ;

{END INSTRUCTION} ;

{MICROP Name} {ASSIGN}
Fieldname=value . . . ;

{INTRINSIC} {'NAME'} ;

II. PROGRAM SECTION

{PROGRAM Name} {WIDTH numberofbits}
[LENGTH numberofwords] ;

[label:] . . . { Fieldname=value / Micropname arguments } . . . ;

{ORG address} ;

{END} ;

III. EITHER DEFINITION OR PROGRAM SECTION

{label:} . . . {EQU value} ;

{label:} . . . {SET value} ;

{LIST} { OFF / SOURCE / SOURCE OBJECT } ;

{OBJECT} { ON / OFF } ;

{SPACE} [value] ;

{EJECT} ;

{TITLE} ['TEXT'] ;

EXTENSIONS TO THE BASIC LANGUAGE

I. EXPRESSIONS

The following is the format of expressions:

operand operator operand

Operators are processed according to the following evaluation hierarchy:

1) SHL, SHR
2) +, –
3) EQ, NE, GT, GE, LT, LE
4) NOT
5) AND
6) OR, XOR

II. The DCL Statement

[(absolutelocation):] [label:] .. {DCL}
[value] [,width] ;

III. Extended MICROP Formats

1. MICROP Micropname [ASSIGN {operand} ...] [DEFAULT {operand} ...] ; in format 1, "Micropname" can be of the following two forms:

2. Micropname=Arg

3. Micropname [(Arg, . . . ,Arg)] [Arg, . . . ,Arg]

Arguments in format 3 are recursive and can be expanded as follows:

4. [Arg] [(Arg, . . . ,Arg)]

Formats 3 and 4 can be expanded to assign each argument a default value:

5. [Arg=default] [(Arg=default, . . . ,Arg=default)]

The operand of format one can have only one of the following formats:

6. Fieldname=value

7. IF booleanvalue THEN operand ELSE operand FI

8. Micropname=value

9. Micropname [(value, . . . ,value)] [value, . . . , value]

Note that format 7 is recursive. Also note that formats 8 and 9 match formats 2 and 3.

Formats 2 and 3 define the microp called and 8 and 9 define the calling microp.

APPENDIX I — MICRO ASSEMBLER ERROR MESSAGES

FATAL ERROR 01

The Micro Assembly Program has insufficient internal work space to assemble the source program. The assembly process is terminated at this point.

FIELD ERROR 01

In the instruction format selected by a Microinstruction statement, a field was not assigned a value and had no default. The field is set to all zeros.

FIELD ERROR 02

(1) More than one value was assigned to a microinstruction field, or (2) fields from conflicting formats were assigned values. (1) The first value assigned is used, or (2) the first format defined in the definition section is used.

FIELD ERROR 03

(1) More than one value was set by Microp DEFAULT to a field, or (2) fields from conflicting formats were set by Microp DEFAULT. Additionally, these defaults were not overridden by a field assign. (1) The first value set is used, or (2) the first format defined is used.

MICROP ERROR 01

In a microp, an argument was not assigned a value and had no default. The argument is assigned the value of zero.

MICROP ERROR 02

In a microp, an argument was given an improper value. This error is specified in a MICROP definition with an IF clause without an ELSE clause.

MICROP ERROR 03

The operand specified in an INTRINSIC statement was not found in the assembler intrinsic file. The statement is ignored.

SYMBOL ERROR 01

A symbol was not defined in the assembly. A value of zero is used.

SYMBOL ERROR 02

A symbol was not previously defined in the assembly. A value of zero is used.

SYMBOL ERROR 03

An improper symbol was used in an expression. A value of zero is used.

SYMBOL ERROR 04

A symbol has been previously defined and may not be redefined by this statement. This definition of the symbol is ignored.

SYMBOL ERROR 05

The micro assembler symbol table is full. This symbol definition is ignored.

SYNTAX ERROR 01

(1) Invalid syntax — an improper sequence of language elements (tokens) was found, (2) Invalid character — a character not in the micro assembler character set was found, or (3) Microp argument error — an invalid argument operand was found. (1) One or more language elements are ignored, (2) the character is ignored, or (3) the argument operand is ignored.

SYNTAX ERROR 02

In a numeric constant, the final character indicating the radix of the constant is invalid. A value of zero is used.

SYNTAX ERROR 03

In a numeric constant, invalid integers have been used for the given radix. A value of zero is used.

SYNTAX ERROR 04

In a character constant, the ending single quote was omitted. A value of zero is used.

SYNTAX ERROR 05

The value of a self-defininig constant exceeds the maximum allowable bit size (specified by the M toggle). A value of zero is used.

SYNTAX ERROR 06

A location counter reference ($) was used improperly in the definition section at the assembly. A value of zero is used.

SYNTAX ERROR 07

The syntax was too complicated to analyze. Probably, too many levels of parentheses were used. One or more language elements are ignored.

VALUE ERROR 01

The location counter has exceeded the PROGRAM length value. High order address bits may be truncated in the object file.

VALUE ERROR 02

The value of an ORG operand exceeds the program length value. High order address bits may be truncated in the object file.

VALUE ERROR 03

The PROGRAM word width may not be zero. A value of 8 bits is used.

VALUE ERROR 04

The combined widths of the fields in a format exceed the width of the format.

VALUE ERROR 05

The value of a width operand is negative or exceeds the maximum bit size (M toggle). A value of one is used.

VALUE ERROR 06

The value of a (1) DCL width or (2) SPACE operand is negative or exceeds the host machine word size (F toggle). The default value is used.

MICRO FORMAT PROGRAM ERROR MESSAGES

FATAL ERROR 01

The Micro Format Program has insufficient internal work space to produce the object output. The micro format program is terminated at this point.

FORMAT ERROR 01

For object output, 2650 format wasn't selected when the LENGTH operand of the MEMORY statement was omitted. Output is suppressed.

FORMAT ERROR 02

For object output, the FROM address of the OUTPUT statement must be zero when the LENGTH operand of the MEMORY statement is omitted. Output is suppressed.

FORMAT ERROR 03

For object output, the WIDTH operand of the MEMORY statement must be less than or equal to eight when 2650 format is selected. Output is suppressed.

FORMAT ERROR 04

For object output, the size of each object output word for the format selected exceeds the record length of the object output file. Output is suppressed.

OBJECT ERROR 01

In the intermediate object file, multiple object words are assigned to the same address. The last object value is used.

OBJECT ERROR 02

The program section name specified in the OF operand of the OUTPUT statement was not found in the intermediate object file. Output is suppressed.

OBJECT ERROR 03

A microinstruction was found in the intermediate object file whose format was not specified with a SELECT statement. The microinstruction is not placed in the object file.

OBJECT ERROR 04

The microinstruction format specified by a SELECT statement was referenced in a previous SELECT statement. The SELECT statement is ignored.

OBJECT ERROR 05

Field names from different microinstruction formats are referenced in a SELECT statement. The first format referenced is used, and the second field name is replaced by a constant zero.

OBJECT ERROR 06

When DCL data occurs in a program section to be output, the MEMORY WIDTH must equal the word width of the program section specified in the PROGRAM statement. The DCL data is not placed in the object output file.

OBJECT ERROR 07

The Micro Format Program symbol table is full. The symbol table is used to store field and program section names from the intermediate object file. Some names may be undefined.

SYNTAX ERROR 01

(1) Invalid syntax — an improper sequence of language elements (tokens) was found. One or more language elements are ignored.

(2) Invalid character — a character not in the Micro Format Program character set was found. The character is ignored.

SYNTAX ERROR 02

In a numeric constant, the final character indicating the radix of the constant is invalid. A value of zero is used.

SYNTAX ERROR 03

In a numeric constant, a character invalid for the radix is used. A value of zero is used.

SYNTAX ERROR 04

In a character constant, the ending single quote was omitted. In this case the first character of the string is used as a single character constant.

SYNTAX ERROR 05

The value of a self-defining constant exceeds the maximum allowable bit size (specified by the M toggle). A value of zero is used.

SYNTAX ERROR 06

The number of bits (select operands) in a SELECT statement does not conform to the MEMORY WIDTH and the instruction format width. The SELECT statement is ignored.

SYNTAX ERROR 07

The syntax was too complicated to analyze. Usually, too many levels of parentheses were used. One or more language elements are ignored.

VALUE ERROR 01

The address range selected by the FROM and TO operands of an OUTPUT statement is larger than the MEMORY LENGTH. Data in higher addresses will not be output.

VALUE ERROR 02

The string operand of a FORMAT statement is not '2650'. The FORMAT statement is ignored.

VALUE ERROR 03

The string values of the BINARY operand of a FORMAT statement contain more than one character. The first character of the string is used.

VALUE ERROR 04

In a SELECT statement, the bit selection constant of a field name exceeds the length of the field. A constant zero is used.

VALUE ERROR 05

The value of a width operand is negative or exceeds the maximum bit size (M toggle). A value of one is used.

VALUE ERROR 06

In a SELECT statement, the value of a constant operand is not zero or one. Zero is used.

VALUE ERROR 07

No object data was output in the address range selected by the FROM and TO operands of an OUTPUT statement. All words of the object output are left unprogrammed (default values).

NOTES

NOTES

NOTES

signetics

a subsidiary of **U.S. Philips Corporation**

Signetics Corporation
P.O. Box 9052
811 East Arques Avenue
Sunnyvale, California 94086
Telephone 408/739-7700

Printed in USA June 1977