# AMDASM/80

Advanced Micro Devices, Inc.

Advanced Micro Devices

# AMDASM™ /80 Reference Manual

## MDS Resident Microassembler

August 1977

# Table of Contents

# List of Figures

# List of Tables

# 1 Chapter 1

## 1.1 Introduction and Purpose

An assembler is a program that “reads” another program written in a symbolic form and produces an output of binary words corresponding to the symbolic input. A microprogram assembler is a special kind of assembler, formally called a “meta-assembler”. AMDASM is a meta-assembler.

An assembler read a program written in a Symbolic language and produces a binary language referred to as machine-level language. The Assembly symbolic language is referred to as assembly language. It is defined on the creation of the assembly language, which is based on specific hardware and can be programmed for many activities. The user in this case does not have any input to the symbolic assembly language nor to the format of the machine language.

A meta-assembler differs from an ordinary assembler in that most of the symbols are defined by the user prior to the assembly. A meta-assembler is specific to the user’s defined hardware. The user also controls the format produced as output by the meta-assembler, which is the “microword”. The microword is customized by the user (designer).

In an ordinary assembler, the user may define the labels for instructions and symbols for particular data words, but the instructions themselves, including the associated word length and format are in general already defined by the assembler that belongs to the hardware being run. This makes sense since the assembler is designed to convert an established set of formats into “machine language” for a particular machine (Am9080A; PDP machines; VAX machines; etc.) When you purchase a computer (such as a PDP or a VAX), the assembler is designed and ships with it. Multiple users would be programming different functions within the limits of the fixed architecture of the machine.

A meta-assembler, called a microprogram assembler or microassembler, however, must be far more flexible than a traditional assembler, since it must be useful for many hardware configurations. Each different hardware configuration may require a different format and may require word length (microwords) over 100 bits in length. A sample is shown below. The number of bits in any field and the number of different fields required is up to the user for the specific design.

**Figure 1-1 From *Bit-Slice Design: Controllers and ALUs*:**

**Machine–Level Instruction**

| OpCode | Destination Register R1 | Source Register R2 |
|---|---|---|
| 15 -------8 | 7 – 4 | 3 – 0 |

**Microword Instruction – bit fields and bit field widths (number of bits) will vary with the design**

| Branch Address | Am2910 INST | CC MUX | IR LD | Am2903 A and B | Am2903 Source | Am2903 ALU | Am2903 Destination | Status Load | Shift MUX | ETC. |
|---|---|---|---|---|---|---|---|---|---|---|
| 16 bits | 4 bits | 1- bit | 1 bit | 8 bits | 16 bits | 4 bits | 16 bits | 1 bit | 1 bit | X bits |

*The microprogram instruction format is under the user’s (designer’s) control and totally depends on the architecture of the hardware to be controlled. The Machine level instruction format is fixed.*

Moreover, in a microassembler, a format rarely establishes the entire contents of the microinstruction, but rather defines only a few bits in the total microword.

These requirements imply that a microprogram assembler must consist of two distinct operations. The first operation is the establishment of word length and the ***definition*** of formats and constants: ***The Definition File***. The second operation is the traditional assembly process (the Assembly File) performed on a program the uses the formats and constants from the Definition File. The microprogram assembler, therefore, differs from the traditional assembler in that it may be configured by the user to accept any word size, formats, and constants the user requires.

The assembler written by Advanced Micro Devices is a very powerful meta-assembler, useful not only with the AMD 2900 family, but with any microprogrammed machine. [DEC had a meta-assembler that could be used in the same way.] These meta-assemblers operate in two phases: the Definition Phase; and the Assembly Phase.

The Assembly Phase executes second and is like any other assembler. It read a symbolic program, handles most common assembler features such as labeling and setting the address counter, and produces a binary output, various listings, and cross-reference tables.

The Definition Phase is executed first to set up the table that associates the user's format names and constant names with their corresponding bit patterns.

The Definition Phase lets the user define symbols for formats (format names), and build the microinstruction word length. In the Definition Phase, the length of the microinstruction is defined first. The word may be any length (1 128 bits). [Today's systems can have complex microwords of over 300 bits in length. This document and program was designed in 1977, pre-personal computers.] This is adequate for all but the most sophisticated processors. [In 1977, the simplex computer design was as far as we had envisioned.]

Each of the user-defined symbols has a specific bit pattern associated with it. [Defined by the part controlled or supplied by the designer of a given module.] A format name is used to define all, or part, of one microinstruction. The format definition may consist of:

- Numeric fields, which are defined to contain specific bit patterns.
- Variables, which will be filled in when the format is invoked.
- "Don't Care" states.

Once the definition Phase has been executed, its output may be retained and used by future programs.

A useful feature of the AMD Meta-assembler is that "don't care" states are retained until defined, which may not happen until after the assembly process, during a third, or post-processing phase. A listing of the microprogram at the conclusion of the assembly phase shows an "X" for every undefined bit. This is useful during the development process before the microword length has been optimized by sharing fields.

Following assembly of the user's program, a file is retained which contains the assembled microprogram. This file is then available for post-processing to create paper tapes [1977] for PROM blowers. [Today's PROMs and EPROMS and other memory units are programmed digitally.] The output utility can select columns and rows for a given PROM tape, freeing the user from any restrictions regarding the organization of the microprogram memory, and simplifying the generation of a new tape for each of the many PROMs in the system.

The program to be assembled may be written using any of the features specified during the Definition Phase. In the simplest case, the Assembly Phase source program might be written using just strings of ones and zeros, with the Definition Phase consisting only of the microinstruction word length. At the other extreme, the Assembly Phase source program may refer to multiple format names from the Definition Phase for each microinstruction. Any number of formats may be overlayed to define a single microinstruction, as long as the defined or variable fields of each format fall into the "don't care" fields of the other formats invoked. A user might define a set of formats specifying sequence control operations, another set for data control, and a third for memory control.

The AMD assembler has been written to maximize its flexibility and ease of use for hardware designers. Every effort has been made to make the program proficient on the machine and efficient at the human interface, with a minimal knowledge of the host machine's operating system required.

## *1.2 Language Comparisons*

### 1.2.1 High-Level Language

High-level languages are fairly free format, i.e., they have few columnar placement restrictions on the coding form (free-form), use pseudo-English mnemonics (LET, GOTO, IF), and have prewritten functions. Their capabilities include arrays, loops, branches and subroutines, with the emphasis on structured programming support with IFTHEN-ELSE, CASE and PROCEDURE statements. High-level language instructions translate into 6 or more machine level instructions, cutting the time to generate the program. Ten high-level commands could translate to sixty machine level instructions. Different languages have different ratios. Different instructions (commands) within the different languages have different ratios of translation.

### 1.2.2 Assembly Level Language

Assembly-level languages have a more restricted format, require a precise data definition, may involve the programmer in program placement in memory, and use mnemonics for instructions but have more of them. Most instructions or statements are restricted to one operation - hence the approximate one to one translation ratio. The assembly level programmer in general must know more about the machine being used than the programmer who writes in FORTRAN or BASIC.

### 1.2.3 Machine Level Language

Machine-level languages are the closest to the system of the software level languages. They are usually written using an encoding of instructions, data and addresses in either octal or hexadecimal notation are more tedious to construct and debug and are more restrictive in the format required than the assembly level languages. They can require more specific detail from the programmer, depending on the complexity of the system being programmed.

### 1.2.4 Microprogramming

The machine level instructions are what the computer control unit (the CCU) receives. In a microprogrammed machine, each machine level instruction (referred to as a macroinstruction) is decoded and a microroutine is addressed which, as it executes, sends the required physical control signals in their proper sequence to the rest of the system. This is where the software instruction via a firmware microprogram is converted into hardware activity. The translation ratio is not predictable.

## 1.3 Definition of Terms

Since there are no standard terms associated with microassemblers, the more common terms used in this manual are listed below.

**Table 1-1 Terms and Definitions**

| Term | Definition |
|---|---|
| Δ | Indicates a required blank character. |
| Name or label | 1-8 characters, which are assigned a value by the programmer or the assembly process. Labels are used only in the Assembly File. |
| Constant | A specific pattern of 1-16 bits. |
| Constant name | A name for a constant. |
| Field | A group of adjacent bits in a microinstruction. |
| Format | A model for a microinstruction consisting of fields which contain constants, variable, and “don’t cares”. |
| Format Name | A name for a format *(recursive definition!)* |
| Line | An input line of up to 128 characters on a console, teletype, a paper tape reader, or a diskette file. *(this is 1977.)* |
| Modifiers | Symbols (*%:-$) which indicate that the data for a given field is to be modified. |
| Attribute | A modifier, which is permanently associated with a field. |
| Designator | A symbol (V, X, B#, Q#, D#, or H#), which indicates the type of field or constant: variable (V). “don’t care” (X), binary (B#), octal (Q#), decimal (D#), or hexadecimal (H#). |
| Delimiters | A symbol (:,/) which indicates the end of a name, the end of a field, or the continuation of a statement on another line, respectively. |
| Default Values | The value, which will be substituted if an explicit value is not specified. |
| Options | Choices available which indicate the input and output devices to be used, the type of output listing desired, and processing of one or both phases (Definition and Assembly). |
| { } | Braces indicate that the enclosed parameter is optional |

## 1.4 Definition Phase

The AMDASM Definition Phase includes the following features:

- A name is a packed group of 1 to 8 characters.
- A name may be assigned to a constant value.
- A name may be used to define a format whose fields are given as variables, “don’t cares”, explicit bit patterns (values), or constant addresses by using appropriate designators.
- Blanks may be used to improve readability.
- Microword length may vary from 1 to 128 bits.
- Modifiers include: inversion, truncation, negation, and designation of a field as an address field to be right-justified (placing a value in a field at the right with leading bits set to zero).
- The ability to set a “page” size via the attribute $. This permits error detection when the Assembly Phase calls for a jump or branch to an address which is on a different “page” of the microcode.

Data from the Definition Phase may be retained for use with subsequent Assembly Phase source programs and/or it may be modified as desired.

## 1.5 Assembly Phase

The Assembly Phase provides for input of the microprogram source statements, conversion of format and constant names to their appropriate bit patterns, substitution of values for variable fields in the format, and generation of listing and binary output. The assembly source program will use references to format names and constant names from the Definition file. It will also contain statements that associate labels with addresses, control assembler operation, and provide program location counter control.

The AMDASM Assembly Phase includes the following features:

- A microword may be assembled by referring to one or more format names from the Definition File.
- A microword whose format was not specified in the Definition File may be specified by using the built-in free-form format command.
- The programmer may control the program location counter to set the origin and/or to reserve storage.
- The programmer may choose one of four different output-listing formats.
- A constant may be defined using values and/or expressions.
- Errors are detected and listed. Severe errors cause processing to halt.

Output of the Assembly Phase is an object file that contains the complete microprogram. Post processors can directly convert this object file to any form needed, such as hexadecimal or BNPF punched on paper tape.

## 1.6 Implementation

AMDASM/80 operates on the Intel Intellec® MDS-DOS System under the ISIS-II© Operating System.[1]

## 1.7 Assembler Operation

AMDASM is placed into execution by control statements from the console input device.

The Definition file is processed and if it contains no errors the Assembly Phase begins. Assembly Pass 1 assigns values to Assembly file labels and allocates storage. Pass 2 translates the Assembly File source program into Object code.

[1] Intel and Intellec are registered trademarks of Intel Corporation.

User-selected options determine whether the Definition Phase is to be executed or if a previous execution of that phase has already established the table of formats in a file that will be used by the Assembly process.

The ISIS-II© operating system allocates all necessary input and output resources, such as files, automatically.

# 2 Chapter 2

## 2.1 *Definition Phase*

The definition phase allows the user to define the microword length, constants, and formats which will be used to write source programs for the target machine.

## 2.2 *Definition File*

The definitions are input via a sequence of instructions called the Definition File whose content includes the following items:

---

TITLE

WORD n

Format definitions

Constant definitions

Assembler control statements

Subformat definitions

Comment statements

END

---

The control statement WORD must appear as the first statement in the Definition file after the optional TITLE statement.[2] The END statement must be the last statement in the Definition File.

The other statements (shown boxed) may be interspersed throughout the body of the file.

To facilitate readability, blanks may appear in most parts of these statements, and an entire blank line mat be inserted by entering a semicolon and a carriage return.

[2] TITLE should not really be optional – poor programming choice if it is.

## *2.3 Assembler Control Statements*

Control statements are used to set microword length, control printing, and indicate the end of the Definition File statements.

### 2.3.1 TITLE

If the user wishes to have a title printed on the Definition File statements, the first statement input should be TITLE. The general form is:

| form: | |
|---|---|
| TITLEΔ | title desired by user |

TITLE must:

- Begin on a new line.
- Be followed by a blank and a maximum of 60 characters.

### 2.3.2 WORD

WORD must be the first statement input by the user after the optional TITLE is given. Its general form is:

| form: |
|---|
| WORDΔ n |

WORD must be followed by a decimal integer value n which indicates the microword size in bits (range 1-128).

WORD must:

- Contain no embedded blanks between the letters of the control statement WORD.
- Be followed by at least one blank and 1 to 3 decimal digits.
- Be the first input line (second input line if TITLE was used).
- Begin on a separate line.

IF WORD is omitted, assembly will halt as the Definition Phase must know the size of the microword to proceed.

### 2.3.3 LIST

LIST indicates that the following statements are to be printed whenever printing of the Definition File input is requested. This feature will be most useful when correcting or modifying a Definition File. (AMDASM selects LIST as the default option. NOLIST must be specified if the user does not want to print the Definition File source statements.) The general form is:

| form: |
|---|
| LIST |

LIST must:

- Begin on a new line.
- Be followed by at least one blank or a carriage return.
- Precede the Definition File statements that are to be printed.
- Be interspersed between complete definition statements.

### 2.3.4 NOLIST

NOLIST turns printing off, and no printing of the Definition File input statements will occur until LIST is encountered. However, any source statement containing an error will still be listed. The general form is:

| form: |
|---|
| NOLIST |

NOLIST must:

- Begin on a new line.
- Be followed by at least one blank or a carriage return.
- Precede the Definition File statements that are not to be printed.
- Be interspersed between complete definition statements.

### 2.3.5 END

END indicates the end of the Definition File. If END is omitted an error message will be printed but processing will continue. The general form is:

| form: |
|---|
| END |

END must:

- Begin on a new line.
- Be the last statement in the Definition File.
- Be followed by at least one blank or a carriage return.

## 2.4 Definition Statements

Definition statements are used to define constants, full-microword formats, or partial microword formats. The general form of thee statements is:

form:

name: definition wordΔ field1, field2, . . . , fieldn

The definition words are:

EQU

DEF

SUB

### 2.4.1 EQU

EQU is used to equate a constant name to a constant value or expression. The general form is:

form:

name: EQUΔ constant (or expression) [; comment]

This equates the characters given in the name position to the value of the constant or expression. Only one expression or constant is permitted following the EQU.

The following set name a “COUNTER” = 1011010.

COUNTER: EQUΔ B#1011010

Future references to the bit pattern 1011010 may be made by using the word COUNTER.

Each EQU must:

- Begin on a new line.
- Begin with a name:
- The name: must be followed by EQUΔ—blanks between the “:” and EQU are optional.
- Contain a constant or expression that represents the bit pattern for one field.
- Define a value that can be represented in 16 bits ($2^{16}$ – 1 maximum).

Each EQU may:

- Be followed by a semicolon and comment after the constant or expression.
- Be continued on additional lines by using / (forward slash) as the first nonblank character in those lines.
- Be used in the Assembly File as well as in the Definition File.

### 2.4.2 DEF

DEF is used to define a complete microword format establishing the contents of unvarying portions of the microword and establishing the position and length of variable and “don’t care” fields. In addition, default values for variable portions of the word may be specified. The general form is:

---

form:

---

name: DEFΔ field1, field2, field3, . . . , fieldn

---

Each DEF must:

- Begin on a new line.
- Be preceded by a name.
- Be followed by one or more blanks, then the fields, separated by commas
- Have the sum of the lengths of all fields exactly equal to the microword length specified by WORD.
- Specify every bit in the microword in terms of ones, zeros, “don’t cares”, or variables.

A DEF may:

- Contain blanks between name: and DEFΔ.
- Contain 1 to 30 fields separated by commas.
- Be continued on additional lines by using / (forward slash) as the first nonblank character in those lines.
- Be followed by a semicolon and comment after any full field is defined.
- Contain (in any field) a subformat name or constant name that has been PREVIOUSLY defined.
- Contain a variable, “don’t care”, constant or expression in any field.
- Contain a variable field that specifies a default value for the field. The default value may be a constant or a “don’t care”.
- Be overlayed on “don’t care” fields with another format to obtain a complete micrtoword d uring the Assembly Phase. Overlaying on other than “don’t care” fields will result in errors, so this feature must be used with care.

### 2.4.3 SUB

SUB is used to define a subformat that is a portion of the microword. A subformat is the same as a format except that it contains fewer bits than the full microword. The fields may be constants, variable, or “don’t cares”. Its gerneral form is:

---

form:

---

name: SUBΔ field1, field2, field3, . . . , fieldn

---

Each SUB must:

- Begin on a new line.
- Be preceded by a name.
- Be followed by one or more blanks, then the fields, separated by commas
- Precede the DEF in which it is first used.

- Not be used in the Assembly file.

A SUB may:

- Be less than a microword length in bits.
- Contain 1 to 10 fields separated by commas.
- Be used as a field in a DEF.
- Be continued on additional lines by using / (forward slash) as the first nonblank character in those lines.
- Be followed by a semicolon and comment after any complete field.
- Contain (for any field) a constant name that was PREVIOUSLY defined, a constant, expression, variable, or “don’t care” specification.

A SUB will be useful when several formats contain identical adjacent fields. IN this case, the subformat name may be used in each DEF whenever these fields occur.

### 2.4.4 Examples of EQU, SUB, DEF

EQU might be written as:

R2: EQUΔ B#010

This defines the name R2 as a 3-bit constant with the bit pattern 010. Whenever the symbol R2 is used, the bit pattern 010 will be substituted

SUB might be written as:

SHFTRT: SUBΔ 3V, B#10110, 5X

This defined SHFTRT as a subformat with a 3-bit variable field (3V), a 5-bit constant field (B#10110), and a 5-bit “don’t care” field (5X), for a total of 13 bits.

DEF might be written as:

ADD: DEFΔ 3V, B#10110, 5X, B#10110, 5X, B#0011, 4X, B#010

This defines ADD as a format with:

- A 3-bit variable field (3V)
- A 5-bit constant field (B#10110)
- A 5-bit “don’t care” field (5X)
- A 5-bit constant field (B#10110)
- A 5-bit “don’t care” field (5X)
- A 4-bit constant field (B#0011)
- A 4-bit “don’t care” field (4X)
- A 3-bit constant field (B#010)

This gives a total microword length of 24 bits.

Alternatively, the same microword could be written using the subformat name SHFTRT and the constant name R2 (previously defined) by writing:

ADD: DEFΔ SHFTRT, B#0011, 4X, R2

## *2.5 Continuation*

Any statement may be continued on additional lines by placing a / (forward slash) as the first nonblank character in those lines.

A continuation must:

- Have a slash as the first nonblank character in its line.
- Preferably be indicated after a complete field (including the comma) has been given on the preceding line.
- Never occur between the designators B, D, Q, or H, and the # sign.

Examples of continuation are:

```
SHFTRT: SUBΔ 3V, B#10110,
/5X
ADD: DEFC 3V, B#10110, 5X,
/B#0011, 4X, B#010
```

## 2.6 Comment Statements

A comment statement is used to provide information about program variables and program flow. The general form is:

| |
|---|
| form: |
| ; comment text |

*Note: for structured microcode, every single line gets a comment at the minimum. This methodology works. Saves time. Aides in debug and test. Aides reviewers.*

A comment may be a full or partial line. The assembler ignores all data from the semicolon to the end of the input line.

Comments must:

- Begin with a semicolon.
- Be placed after a complete field if used within a DEF or SUB statement, in which case subsequent fields for the DEF or SUB must begin on a new line with a / (forward slash) indicating that they are a continuation of this DEF or SUB.

For example:

1. SHFTRT: SUBΔ 3V, ; this is a shift-right statement
2. /B# 10110, 5X; which is continued on a second line
3. ; the ADD given below is a complete microword format
4. ADD: DEFΔ SHFTRT, B#001, 4X, R2
5. ; Total number of bits for SHFTRT is 13
6. ; the bit pattern for SHFTRT will be substituted
7. ; in the ADD given above

Statements 3, 5, 6, and 7 are full comment lines. Statements 1 and 2 are statements to be processed but all characters after the ‘semicolon’ will be treated as comments. The SUB begun in statement 1 is continued in statement 2 where “/” indicates continuation.

## 2.7 Names

Names may be user-defined constant names, format names, or subformat names.

Names must:

- Be the first element in a statement.
- Begin with an alphabetic character (A-Z) or a period (.).
- Be terminated by a colon (:).
- Contain a maximum of 8 characters not including the colon.
- Not contain any embedded blanks.

- Be followed by EQU, DEF, or SUB.
- Contain only alphabetic characters (A-Z), a period (.), or the digits 0 through 9 in positions 2 through 8.

Names may:

- Contain more than 8 characters but will be truncated after the first 8 characters.
- Be preceded by blanks.
- Be followed by blanks after the : and before the EQU, SUB, or DEF.

Examples of proper names are:

NUMBER:

.SHIFT:

REG.3:

Improper names are:

| | |
|---|---|
| *ADD | special character used |
| SHIFT LEFT: | embedded blank, more than 8 characters |
| 3MUXCNTL: | first character not A through Z or period |

## 2.8 Designators

A designator is used to indicate the type of constant or field being defined. Designators are:

**Table 2-1 Designators and their Definitions**

| Designator | Definition |
|---|---|
| V | A variable field. V must be preceded by decimal digit(s) giving an explicit length for this (i.e., the bit length). |
| X | A “don’t care field. X must be preceded by decimal digit(s) giving an explicit length for this field (i.e., the bit length). |
| B# | A constant or field whose contents will be represented using binary digits (0 and 1) |
| Q# | A constant or field whose contents will be represented using octal digits (0 through 7) |
| D# | A constant or field whose contents will be represented using decimal digits (0 through 9). A D# must be preceded by decimal digit(s) giving an explicit length (number of bits) when representing a field. |
| H# | A constant or field whose contents will be represented using hexadecimal digits (0 through 9, A through F) |

The designators B#, Q#, D#, H# must have NO BLANKS between the letter and the pound sign (#). When used after nV, these designators indicate that this variable field will default to this type unless a designator is given for this field during the Assembly Phase. For example, if all variable fields are given as nVQ# in the Definition Phrase, all values for these variable fields that are octal may be written during the Assembly Phase by writing only the necessary octal digits.

## 2.9 Examples of Designators

A DEF is used to associate bit patterns with a symbol (format name). One example is:

```
ADD: DEFΔ 4VH#, Q#7, B#01, 8X, 4D#12
```

ADD defines a microword format where:

- Field 1 is a variable field with an explicit length of 4 bits with a default type (but no default value) of hexadecimal.
- Field 2 is a variable field with an implicit length of 3 bits containing the value of the octal digit 7 (bit pattern 111).
- Field 3 is a constant field with an implicit bit length of 2, containing the value of the binary digits 01 (bit pattern 01).
- Field 4 is a "don't care" field with an explicit length of 8 bits. The bit pattern may be 8 zeros or 8 ones (or any 8-bit binary mix) as this field is not used by this format.
- Field 5 is a constant field with an explicit length of 4 bits containing the value of the decimal digits 12 (1100).

AN EQU is used to associate a bit pattern with a symbol (constant name). One example is:

```
TWOK: EQUΔ 2048
```

This assigns the bit pattern 100000000000 and a length of 12 bits to the name TWOK. The 2048 is assumed to be decimal (no reason given) and the length is taken from the rightmost bit through the leftmost bit in which a 1 appears.

Thus,

```
EIGHT: EQUΔ 8
```

Yields a bit pattern 1000 with a length of 4.

```
SIX: EQUΔ 6
```

Yields the bit pattern 110 with a length of 3.

Alternatively, by using different designators, the constant:

```
TWOK: EQUΔ 2048
```

Could be written:

```
TWOK: EQUΔ B#100000000000
TWOK: EQUΔ Q#4000
TWOK: EQUΔ H#800
```

All of these yield the bit pattern 100000000000 and a length of 12.

When a designator B#, Q#, D#, or H# is given after a V, it becomes a permanent attribute of that field and the assembler assumes that any value specified for that field will be given in digits appropriate to the designator chosen.

These permanent designators for variable fields may be overridden when using the format during the Assembly Phase.

*Note: If a variable field has no designator given, it defaults to binary.*

Structured code requires that the designator be given.

## 2.10 Fields

Each field following a definition word must:

- Be followed by a comma unless it is the last field in a format or subformat.
- Define a constant field using the designators B#, Q#, H#, or D# and the appropriate digits.

or

- Be a variable that gives a bit length and the designator V.

or

- Be a “don’t care” that gives a bit length and the designator X.

or

- Be a constant name or subformat name that has been previously declared.
- Contain a maximum of 16 bits unless it is a “don’t care” field.

Each field may be given an implicit or explicit length. An explicit length is indicated for a field by using decimal digit(s) before the designator.

Thus,

```
3B#101
```

Indicates a field with an explicit length of 3 bits.

Decimal, variable, or “don’t care” designators require an explicit length before the designator D#, V, or X.

“Don’t care” and variable fields require an explicit length since they do not always contain a definite bit pattern.

Decimal fields require an explicit length since there is no direct correlation between the number of decimal digits given and the number of binary bits desired for this field.

**Table 2-2 Example Field Definitions**

| Example | Description |
|---|---|
| 4V | Defines a variable field with the explicit length of 4 bits. |
| 5D#16 | Defines a constant field with the explicit length of 5 bits and the bit pattern 10000 |
| B#10000 | Defines a constant field with the implicit length of 5 bits and the bit pattern 10000 |

## 2.11 Constants

Constants are used to define fields or associate names with a specific bit pattern.

An example of a constant used to associate a name with a specific bit pattern is:

```
A: EQUΔ 3
```

Further references to the constant name A yields the bit pattern 11.

AN example of a constant used to define fields in a format name is:

```
R: DEFΔ 3B#011, H#7
```

This defines field 1 in the format named R as a constant (explicit length of 3 bits) field with the bit pattern 011. Field 2 is a constant (implicit length of 4 bits) field with the bit pattern 0111.

Alternatively, the R format could be written as:

R: DEFΔ 3Q#3, B#0111

To yield the same bit pattern as before for each field. Field 1 has an explicit length of 3 bits, while field 2 has an implicit length of 4 bits.

## 2.12 Modifiers

Modifiers are place after a constant or after the designator V. After a constant they are used only to alter the value given. When used after V, the modifiers are called attributes of that field and are permanently associated with that field. Attributes will modify any default value given with the variable field in the Definition file and they will modify any value substituted for this variable field when the format name is used in the Assembly File.

Permitted modifiers and their actions are:

**Table 2-3 Modifiers and their Actions**

| Modifier | Action Performed on Preceding Constant |
|---|---|
| * | Inversion (One's complement) |
| – | Negate the number (two's complement) |
| : | Truncate to the left to make the value given fit into the number of explicit bits for this field. |
| % | This field is to be considered an address field. Any value given is to be right justified in the field and any bits remaining on the left are to be filled with zeros. |
| $ | The field is treated as an address within a "paged" memory organization. This attribute permits substitution in this regard and initiates out-of-bounds page checking logic. Used only with variable fields as an attribute (may not follow a default value). |

Examples of correct use of modifiers with constants:

**Table 2-4 Correct Use of Modifiers with Constants**

| Example | Description |
|---|---|
| B#101 * | Yields bit pattern 010 (101 is inverted) |
| 4D#5– | Yields bit pattern 1011 (5 is two's complemented) |
| 6Q#357: | Yields bit pattern 101 111 (The left bits 011 (3) are truncated, 5 and 7 remain. |
| 12H#A5% | Yields bit pattern 0000 1010 0101 (A5 is right-justified in a 12-bit field). |
| 4B#101 | Explicit length is 4 bits, only 3 bits follow the B# but no % sign (indicating right justification) is given. ERROR |
| 5Q#34 | Explicit length is 5 bits, but the 34 generates 6 bits and no ":" has been given to indicate that the leftmost bit is to be truncated. ERROR |

Modifiers must:

- Appear after the value of a constant (i.e., 12H#4C% or 5Q#37:).
- Appear after the V but before the (optional) default value for a variable field (12V%Q#46) it they are to be permanent attributes of the field. The % and Q# become permanent attributes of this variable and are also modifiers of the default value. To modify only the default value, modifiers must follow the value (12VQ#46%).
- Not appear with "don't cares" (e.g., 3X% is ILLEGAL.)
- The modifiers * and – may not both be used for a field.

## *2.13 Modifiers as Attributes*

Modifiers used in variable fields immediately following the V designator are permanent attributes of that field. Thus, 12V%*: indicates a 12-bit variable field and any value givein the Assembly File will be inverted, then right-justified if the value to be substituted is less than the field length, or have the left bits truncated if the value to be substituted is larger than the specified length of the field.

### 2.13.1 Order of Precedence

Modifiers or attributes may appear in any order but will always be processed in the following order:

(*) Inversion or (–) negation

(%) Right justification

(:) Truncation

### 2.13.2 Radix Base Defaults to Binary

Variable fields also use the B#, Q#, D#, and H# as attributes. Once given, the B#, Q#, D#, and H# are permanently associated with that variable field unless overridden. If a variable field has no radix base is specified, it will default to binary.

If the user always wants to input assembly variables in octal, each variable field in the Definition Phase should be written as nVQ#. Then, in the Assembly Phase, the value for this field may be given as 27 and the program will assume that these are octal digits. [Not *assume* – it *knows* because you told it.]

If, in the Assembly File, octal is not desired, the field in the Assembly File program could be written a B#101111, or H#27, etc., to override the octal attribute set in the Definition File.

### 2.13.3 Precautions for H#

The attribute H#, if given with a variable field in the Definition File, may need to be repeated in the Assembly File. This is necessary since the program cannot distinguish hexadecimal values that begin with A through F from names, which may also begin with the letters A through F.

H# may appear as a permanent attribute for a given variable, in which case, the digits 39 could be interpreted as hexadecimal when encountered in the Assembly Phase. However, whenever a value is encountered in the Assembly Phase where the variable definition includes the implicit hexadecimal type (e.g., 12VH#) and the initial digit is one of the letters "A" through "F", the value must be preceded by the H# modifier (e.g., H#BAD) to distinguish it from a symbol of the name (e.g., BAD:EQUΔ123).

## *2.14 Attribute $ ("paging")*

An attribute which may be used only with variable fields is the $, which indicates paged addressing.

When the $ is given with a variable field, the % and the : attributes are automatically set for that field.

The $ will indicate that this is a field whose remaining upper (left most) bits are to be compared with the corresponding bits of the program counter (PC) after the lower (right most) bits have been substituted into the variable (i.e., the truncated bits of this field are compared with the corresponding bits of the PC).

If the truncated bits do not agree with the corresponding bits of the PC, an error occurs.

The desired length of the “page” is determined by the number of bits given as the width of this variable field.

Thus, if a “page” is to be 256 words deep, the variable field would be defined as 8V$. Any substitution for this field would be truncated on the left and the remaining eight right-hand bits will be used for the address. If the truncated left bits do not agree with the corresponding bits of the current program counter value, the substitution would attempt to produce a jump to another page; thus an error message is generated.

For example, if the Definition File contains

```
                    JSR: 3X, 8V$, H#6B
```

And the Assembly File is:

```
                    ORG 0
                    JSR BEGIN
                      *
                      *
                      *
                    ORG 256
          BEGIN:    ADD
```

An error is generated since BEGIN = $256_{10}$ = 1 0000 $0000_2$ while the PC at JSR is 0 0000 $0000_2$ (The left bits that are truncated do not agree).

If the “page” size is 1024 microwords, a variable address field of 10 bits should be used.

IF any label is substituted for this variable, the truncated left bits of the label are compared with the left bits of the program counter to ensure that this label is on the same “page” as the microword.

Examples of the correct use of $:

```
          8V$
          8V$*
          8V*$
          8V$Q#
          8V$Q#7
```

## *2.15 “Don’t Cares”*

A “don’t care” is used to indicate the bits (a field) whose state (bit pattern) is irrelevant in the microword instruction in which it appears.

| |
|---|
| form: |
| nX |

where:

n is the number of bits (in decimal)

X indicates “don’t care”

“Don’t cares”:

- Are set to zero during the Definition and Assembly phase.
- May be changed to ones after the Assembly Phase by post-processing.
- Are the only fields of a DEF that may be overlayed when two format names (DEFs) are combined in the Assembly Phase to form a complete microword.
- Are the only fields that may be greater than the 16-bit field length limit.

## *2.16 Variables*

Variables are used to define microword fields whose contents need not be assigned until assembly time. A variable field may be assigned a default value in the Definition File. The general forms are:

---

form:

---

nV

nV attributes

nV attributes default-value

nV attributes default-value optional-modifiers

nV default-value optional-modifiers

---

A variable field must:

- Be preceded by an explicit length (n) that gives (in decimal) the bit length of the field. ($n \leq 16$)
- Contain V after the length.
- End with a comma (,) if another field follows it.
- Contain a % after the V if an expression or $ is used as a substitute for this field in the Assembly File.

A variable field may:

- Contain attributes (immediately after the V), such as inversion (*), which will always invert any value given for this field.
- Contain a default value given in binary (B#), octal (Q#), hexadecimal (H#), or decimal (D#) followed by the desired digits.
- Contain a designator given with or without a default value which will automatically determine the default type for this field.
- Contain modifiers after the default value. These modify only the default value and are not permanently associated with this variable field.
- Contain a default value given as X (indicating “don’t care”) if the user wishes to overly this field during the Assembly Phase.
- Contain either a default value of “don’t care” or an explicit default value (bit pattern) but not both.

Examples of the correct use of variable fields with a default value of “don’t care” are:

```
3VX
3V*X
3V$X
3V*$X
```

### 2.16.1 Correct Variables

Examples of variable fields are:

**Table 2-5 Correct Field Content and Meaning**

| Field Content | Meaning |
|---|---|
| 3V | A 3-bit field. The contents are variable and will be supplied when this format name is used in the Assembly File. The field type defaults to binary. |
| 3VQ# | A 3-bit field whose contents are are variable. The contents will be supplied when the format name is used during the Assembly File. |
| 3V*% | |
| 3VQ#5 | |
| 3VQ#5* | |
| 3V*Q#5 | |
| 3V*Q#5* | Yields a 3-bit variable field with a default value of 5, inverted, then inverted again by the * following the V. The resulting bit pattern is 101. All values substituted for this field in the Assembly File will be inverted. |

To summarize, attributes places immediately after the V are permanently attached to this field and will operate on any default value given with the field as well as any value substituted for the field in the Assembly File.

Modifiers placed after the default value apply only to the default value.

Attributes given after a V (except % and :) may be overridden for the field by placing a modifier after the value substituted for the field in the Assembly File.

### 2.16.2 Incorrect Variables

**Table 2-6 Incorrect Field Content and Meaning**

| Field Content | Meaning |
|---|---|
| 3VH#7 | The H#7 yields 4 bits. No : was given to indicate that he left bit should be truncated to fit the 3-bit field. |
| 3:VH#7 | The : is in an incorrect position. I should be 3V:H#7 or 3VH#7: depending on wether the truncation is a permanent field attribute or a modifier of the default value H#7). |

In short, attributes must be placed immediately after the V. Modifiers must be placed immediately after the digits given for the default value.

## 2.17 Definition File – Reserved Words

The following words are used during the assembly phase as assembler control statements and may not be used as format names or constant names in the Definition File.

| | | | |
|---|---|---|---|
| ALIGN | EQU | NOLIST | SPACE |
| EJECT | FF | ORG | TITLE |
| END | LIST | RES | |

## 2.18 Definition Phase Error Messages and Interpretations

Processing of a statement is halted when any error in that statement is detected. The next statement is then processed and checked for errors.

AMDASM makes every effort to exactly pinpoint errors; however, certain mistakes in the Definition File such as missing comma between fields would distort the meaning for that statement, and meaningless error messages would occur should further processing be attempted.

The user may get AMDASM or ISIS© error messages. ISIS-II© messages will have the form:

ERROR nn PC nnn

And may be interpreted using the ISIS-II© manual.

AMMDASM errors will have the form:

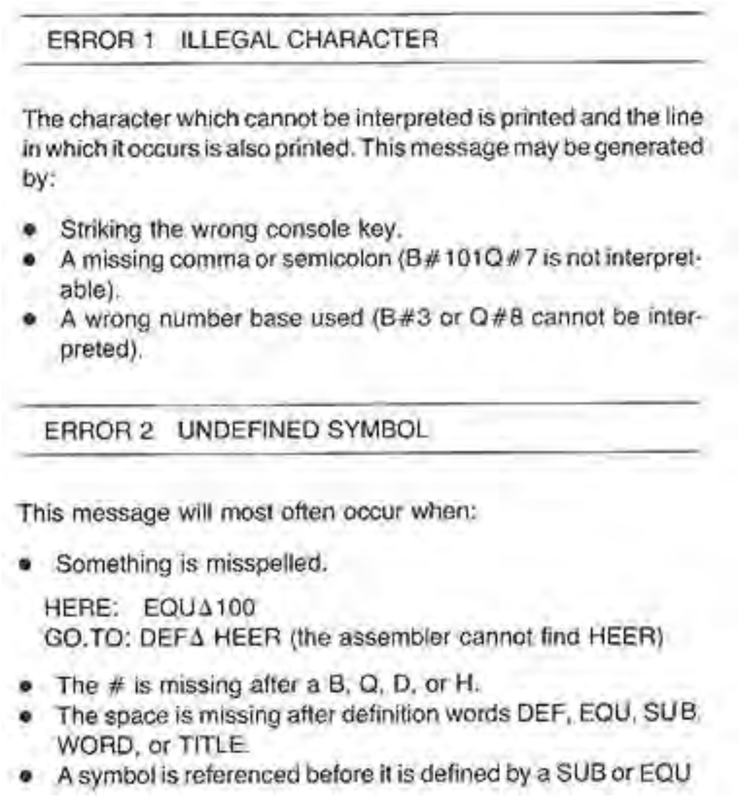
***ERROR n {y}

where:

n is the error number

y, if present, contains an illegal character or symbol.

Errors where n ≥ 100 halt execution. They are listed in Chapter III.



ERROR 1 ILLEGAL CHARACTER

The character which cannot be interpreted is printed and the line in which it occurs is also printed. This message may be generated by:

- Striking the wrong console key.
- A missing comma or semicolon (B#101Q#7 is not interpretable).
- A wrong number base used (B#3 or Q#8 cannot be interpreted).

ERROR 2 UNDEFINED SYMBOL

This message will most often occur when:

- Something is misspelled.

  HERE: EQU∆100
  GO.TO: DEF∆ HEER (the assembler cannot find HEER)

- The # is missing after a B, Q, D, or H.
- The space is missing after definition words DEF, EQU, SUB WORD, or TITLE.
- A symbol is referenced before it is defined by a SUB or EQU

## ERROR 4 DUPLICATE FORMAT

The name given before a format (DEF) has already been used as a name. If names contain more than 8 characters, the first 8 must be unique. Check for misspelled names.

## ERROR 6 DUPLICATE SUBDEFINE

The name given preceding a subformat (SUB) has already been used as a name. If names contain more than 8 characters, the first 8 must be unique. Check for misspelled names.

## ERROR 7 FORMAT FIELD OVERFLOW

The user is permitted a maximum of 30 fields per format name (DEF). This number has been exceeded. The format must be revised and fields must be combined.

## ERROR 8 SUBDEFINE FIELD OVERFLOW

The user is permitted a maximum of 10 fields per subformat name (SUB). This number has been exceeded. Revise the subformat and combine fields or use two subformats for this bit pattern.

## ERROR 9 UNDEFINED DIRECTIVE

No name: was found and the characters given are not TITLE, WORD, LIST, NOLIST, or END.

Check for a missing colon after a name, or misspelling, or blanks in TITLE, WORD, LIST, NOLIST, or END.

## ERROR 10 ILLEGAL MICROWORD LENGTH

Each time DEF is encountered, the assembler checks to see if the sum of the bits for all fields for this format name **exactly** equals the microword length.

Thus, the user is assured that each DEF contains an exact number of bits. If the number of bits in this format does not **exactly** equal the number of bits given with WORD, the interpretation of the faulty DEF is bypassed and the assembler attempts interpretation of the next Definition File instruction.

## ERROR 11 ILLEGAL FIELD LENGTH

The value calculated for this field has a value that cannot be represented by 16 bits. A missing comma may cause the assembler to assume that data (intended to be a field) is an explicit length for the next field.

## ERROR 12 DON'T CARE FIELD TOO LONG.

The explicit length given for a "don't care" field exceeds the microword length specified by WORD. Improper digits may have been assumed for the explicit length due to a missing comma or designator.

## ERROR 14 ATTRIBUTE ERROR

Both the negative (−) sign and inversion (∗) have been assigned to a single variable or constant. This is not permitted. 4V−∗ or 4B#1011∗− are meaningless.

## ERROR 16 MISSING END STATEMENT

The Definition or Assembly File is missing the END statement.

## ERROR 17 ILLEGAL SYMBOL

A character other than A through Z, digits 0 through 9, or period was used in a name, or a comma may be missing between fields.

## ERROR 20 FIELD LENGTH CONFLICT

The calculated or implicit field length for the constant or expression given after the designator does not have the same number of bits as were given by the explicit field length. Check for a missing # or :, or a comma missing after the previous field.

This message may be output when commas are left out. For example,

```
8H#A39Q#274
```

is missing the comma between 3 and 9. Thus the program assumes A39 is to be substituted into the 8-bit hexadecimal field.

Similarly,

```
8H#A3, 9Q27, 4
```

will generate this error message since the comma between the 7 and the 4 is misplaced.

## ERROR 23 MISSING DESIGNATOR

A field has been found which contains only decimal numbers. This is not permitted for a field in a DEF or SUB. Decimal numbers must be input as, n D# digits, where n is the explicit length of the field and digits are the decimal integers which generate the desired bit pattern or field value.

# 3 Chapter III

## 3.1 *Assembly Phase*

The Assembly Phase reads in the source program statements, assigns values to labels and constants, then translates the source program's executable statements into a binary format. The Definition Phase output (a table of format and constant names and their associated bit patterns) is used for this translation.

The user must input the instructions in the order in which they are to be executed. The user may allocate blocks of storage, control printing, and set the program counter via non-executable assembler control instructions which are interspersed with the executable statements.

The allowable Assembly Phase statements are as follows:

| | |
|---|---|
| TITLE<br>LIST<br>NOLIST<br>SPACE<br>EJECT | Assembly control words for printing. |
| ORG<br>RES<br>ALIGN | Assembly control word for program counter control. |
| EQU | Definition word for defining constants. |
| FF | Free form definition word to establish a microword. |
| Executable Statements | References to format names from the Definition Phase. |
| Comments | Used for documentation and program flow. |
| END | End of the Assembly File. |

In this list, only the END statement needs to be placed in a particular position (the last statement of the Assembly File). All other items may be given (and repeated) in any order the user desires. However, with TITLE, only the latest title will be printed at the top of subsequent tables.

## 3.2 Assembly File Statements

Each statement contains an optional label followed by a control word, definition word, or format name. Some control words, definition words and format names must be followed by a value that may be a constant, a constant name, or an expression.

The general form of all Assembly File statements is:

---

form:

---

|  |  |  |
|---|---|---|
| label: Control Word | $\left\{\right.$ | Constant |
| Definition Word |  | Expression |
| FF |  | Constant Name |
| or |  |  |
| Format Name |  | Variable Field Substitute (VFS) |

---

The Assembly File uses five general types of statements. These are listed below with their permissible control words.

- Assembler control statements (LIST, NOLIST, SPACE, EJECT, TITLE, END).
- Location counter control statements (RES, ORG, ALIGN)
- Constant definition statement (EQU).
- Executable instruction statements (format names from the Definition Phase, FF)
- Comment statements (; comment)

## 3.3 Assembler Control Statements

### 3.3.1 TITLE

All data input on the line after TITLEΔ will be printed at the top of each page of output. A maximum of 60 characters may be input for a title. When a new TITLEΔ is encountered the list device moves to the top of the form and succeeding pages will contain this title. The general form is:

---

form:

---

TITLEΔ title desired by user (alphanumeric data to be printed at the top of the page)

---

TITLE must:

- Begin on a new line.
- Be followed by a blank and a maximum of 60 characters.

### 3.3.2 LIST

LIST indicates that the following statements are to be printed whenever printing the of the Assembly File input is requested. This feature will be most useful when correcting or modifying an Assembly File. (Amdasm automatically prints the source statements unless NOLIST is specified by the user.) The general form is:

| |
|---|
| form: |
| LIST |

LIST must:

- Begin on a new line.
- Be followed by at least one blank or a carriage return.
- Precede the Assembly File statements that are to be printed.
- Be interspersed between complete Assembly statements.

### 3.3.3 NOLIST

| |
|---|
| form: |
| NOLIST |

NOLIST must:

- Begin on a new line.
- Be followed by at least one blank or a carriage return.
- Precede the Assembly File statements that are not to be printed.
- Be interspersed between complete Assembly statements.

### 3.3.4 SPACE

SPACE indicates that the assembler is to leave n blank lines before printing the next source statement. The general form is:

| |
|---|
| form: |
| SPACEΔ n |

SPACE must:

- Begin on a new line.
- Be followed by a Δ and a decimal digit.
- Be inserted in the Assembly File at the point where the spaces are desired.

### 3.3.5 EJECT

When EJECT is encountered, the assembler generates blank spaces on a list device so that previous date plus the blank lines equals the specified page length (default is 66 lines). It then begins a new “page”, headed with the title. On a printer a new page is ejected. The general form is:

| form: |
|---|
| EJECT |

EJECT must:

- Begin on a new line.
- Be followed by at least one blank or a carriage return.

### 3.3.6 END

END indicates that the Assembly File is complete and should be processed. The general form is:

| form: |
|---|
| END |

END must:

- Begin on a new line.
- Be the last statement in the Assembly File.
- Be followed by at least one blank or a carriage return.

## *3.4 Program Control Statements*

### 3.4.1 ORG

ORG is used to set the program counter to the value given as n. The general form is:

| form: |
|---|
| ORGΔ n |

ORG must:

- Be followed by at least one blank or a carriage return.
- Have n specified using decimal digits unless one of the designators B#, Q#, or H# precedes the digits given.
- Be used only for setting the program counter forward.
- Be greater than or equal to the current value of the program counter.

ORG may:

- Contain an expression instead of n.
- Be used an unlimited number of times in the Assembly File.

### 3.4.2 RES

RES is used to reserve n words of memory. This increments the program counter by n. The general form is:

| form: |
|---|
| RESΔ n |

RES must:

- Be followed by at least one blank and n.
- Have n specified using decimal digits unless one of the designators B#, Q#, or H# precedes the digits given.

RES may:

- Contain an expression instead of n.
- Be used an unlimited number of times in the Assembly File.

### 3.4.3 ALIGN

ALIGN is used to set the program counter to the next value, which is an integral multiple of the value n. It is used to align the program counter to a specific boundary such that the next microinstruction will be assembled at the next address, which is, for example, an integral multiple of 2, 4, 8, or 16. The general form is:

| form: |
|---|
| ALIGNΔ n |

ALIGN must:

- Be followed by at least one blank and n.
- Have n specified using decimal digits unless one of the designators B#, Q#, or H# precedes the digits given.

ALIGN may:

- Contain an expression instead of n.
- Be used an unlimited number of times in the Assembly File.

## 3.5 Constant Definition Statement

### 3.5.1 EQU

EQU is used to equate a constant name to a constant value or expression. The general form is:

form:

name: EQUΔ constant (or expression) [; comment]

This equates the characters given in the name position to the value of the constant or expression. Only one expression or constant is permitted following the EQU.

COUNTER: EQUΔ B#1011010

sets a name “COUNTER” = 1011010 and future references to the bit pattern 1011010 may be made by using the word COUNTER.

Each EQU must:

- Begin on a new line.
- Begin with a name:
- The name: must be followed by EQUΔ—blanks between the “:” and EQU are optional.
- Contain a constant or expression that represents the bit pattern for one field.
- Define a value that can be represented in 16 bits ($2^{16}$ – 1 maximum).

Each EQU may:

- Be followed by a semicolon and comment after the constant or expression.
- Be continued on additional lines by using / (forward slash) as the first nonblank character in those lines.
- Be used in the Assembly File as well as in the Definition File.
- Be equated to the current value of the program counter by using $ as the designator. The $ may be part of the expression.

## 3.6 Executable Instruction Statements

Executable instruction statements form the body of the assembly phase program. When assembled and with the appropriate substitution of parameters, they form the binary output code of the assembly phase.

### 3.6.1 Executable Instructions Using Format Names

Most executable instructions will refer to the format names established by the definition phase. Their general form is:

form:

{label:}format nameΔ VFS, VFS, {&format nameΔ  VFS, VFS . . . }

{VFS = variable field substitution) (& = overlay)

These formats may be used singly (with appropriate VFSs) or they may be combined (overlayed) with other formats (and their appropriate VFSs. All cases result in the formation of a complete microword.

Executable Instruction Statements must:

- Begin on a new line.
- Contain a format name from the Definition Phase.
- Substitute a constant name, a label, a constant, or an expression for each variable field and these must be separated by commas. If a default value was given in the Definition Phase and is used, the VFS may be omitted.
- Contain & after all VFSs for this format name if it is to be overlayed. The format name (and its VFSs) to be overlayed follows the &.

Executable Instruction Statements may:

- Contain a single format name or may contain an unlimited number of format names to be overlayed.
- Contain the current value of the program counter as the value for a field if $ is the VFS used for that field. The $ may be part of an expression ($ + n) given for a VFS.

## 3.6.2 Free Format Statement FF

Instruction formats, which were not defined in the Definition Phase, may be defined in the Assembly Phase by using the built-in free format command FF. The general form is:

form:

```
{label:} FFΔ   field1, field2, . . . , fieldn
```

An Assembly File may contain an unlimited number of FFs.

Each FF must:

- Begin on a new line.
- Contain a / (slash) as the first non-blank character if continued on another line.
- Contain a maximum of 30 fields.
- Have fields separated by commas.
- Have an explicit length given for “don’t care” fields (nX) or for fields defined using decimal (nD#n).
- Not contain a variable field.
- Not contain a constant name for a field unless that constant has been previously defined in the Assembly or Definition file.
- Not be overlayed with another format name.

Each FF may:

- Be preceded by a label:.
- Contain an expression for any field but the expression must be enclosed in parenthesis and must be preceded by the field length “n”. For example:
  - FFΔ5X, 10 ($-5), B#101
- The value of the expression is automatically right-justified in a field. However, if the value is larger than the field, an error is generated unless the truncation “:” followed the “)” for this expression.
- Contain a field that has the value equal to the current value of the program counter, by using $ for that field or using an expression containing $.

For example, if the contents:

```
WORDΔ 48
AZ:     EQUΔB#01
RB:     EQUΔQ#10
```

Were defined in the Definition File, then the Assembly File could contain the following statements:

```
C: EQUΔ H#C
XTRA: FFΔ 12H#3%, AZ, 18X, C, B#10111
/1X, RB
```

The microinstruction (binary output) for this FF is:

```
0000 0000 0011   01   XX XXXX XXXX XXXX XXXX   1100     10111       X   00100
   121H#3%      AZ                  18X                    C      B#10111      1X    RB
```

Which will be printed in the following format:

```
00000000001101XX XXXXXXXXXXXXXXXX 110010111X00100
```

## 3.7 Overlaying Formats

Formats may be overlayed (combined) with other formats provided that:

- Each bit of format name (#2) that contains a one, or zero, or is part of a variable field must have that bit specified as a "don't care" in the format name (#1) to be overlayed. Subsequent overlays must be on the "don't care" fields remaining after the overlay of all preceding formats.
- Each format is a full microword in length.

For example, it the Definition file contains:

```
ADD:    DEFΔ 5X, 8H#A2, 3X
REG1:   DEFΔ B#0001, 11X
CARRY: DEFΔ 15X, B#1
```

The formats yielded are:

| Format Name | Format | | |
|---|---|---|---|
| ADD | XXXX | 10100010 | XXX |
| REG1 | 0001 | XXXXXXXX | XXX |
| CARRY | XXXX | XXXXXXXX | XX1 |

Then in the Assembly Phase

```
ADDREG:         ADD & REG1
```

Yields

```
00001     10100010   XXX
```

While

```
ADRGCY:         ADD & REG1 & CARRY
```

Yields

```
00001      10100010  XX1
```

### 3.8 Continuation

Any statement may be continued on additional lines by placing a / (slash) as the first nonblank character in those lines.

### 3.9 Comment Statements

Comment statements are non-executable statements, which are used to provide information about the program variables or the program flow. A comment may be a full line or may follow, for example, a constant definition statement. All characters from the semicolon to the end of the input line are not processed and serve merely as a documentation aid. The general form is:

---

form:

---

 ; comment text

---

*Note: for structured microcode, every single line gets a comment at the minimum. This methodology works. Saves time. Aides in debug and test. Aides reviewers.*

A comment may be a full or partial line. The assembler ignores all data from the semicolon to the end of the input line.

Comments must:

- Begin with a semicolon.
- Be placed after a complete field.

### 3.10 Labels or Names

Labels are  names are packed groups of letters and/or symbols which have an associated value. The general form is:

---

form:

---

name:    definition word

       or

label:    format name

---

A name or label's value is determined by the statement type that follows it. Thus,

    Name:  EQUΔ n

Equates the symbol "name" with the value "n".

While

    Label: format name Δ VFS, VFS . . .

Equates to the current value of the program counter, so that reference may be made to this location in the microcode by using this label.

A label or name must:

- Begin with an alphabetic character (A through Z) or period (.) .
- End with a colon.

- Contain no more than 8 characters, exclusive of the colon. (Excess characters are truncated on the right.)
- Contain no imbedded blanks.
- Each be unique. If duplicates are given, the value given at the first occurrence is used and a warning message is issued for each duplicate.
- Not be a reserved word.
- Bu used only with:

  EQUs

  FFs

  Executable instruction statements

When a name is defined by an EQU, the EQU source statement must precede any references to that name. Thus, if CAT: EQUΔ DOG is used and DOG is defined as:

DOG: EQUΔ [value]

Then both EQUs must precede the use of the symbol CAT. A good general rule is to place all EQUs at the beginning of the Assembly File program.

## 3.11 Entry Point Symbols

When generating Mapping PROMs the user may wish to easily obtain the program (location) counter associated with certain portions of the microcode. These program counter value are often referred to as entry points.

Entry points are indicated in the assembly source file as:

Label: : format name Δ VFS, . . .

Except for the double colon, entry points are subject to all the rules applicable to labels.

A list of entry point (symbols and values) may be obtained when AMDASM is executed by requesting the MAP option (see Chapter 4).

## 3.12 Constants

Constants are used with the commands EQU, ALIGN, RES, SPACE, ORG, or as variable field substitutes (VFSs).

They may be expressed in the following manner:

| Form | Permissible Digits | Meaning |
|---|---|---|
| n | 0 through 9 | Decimal Value (default form) |
| B#n | 0 or 1 | Binary Value |
| Q#n | 0 through 7 | Octal Value |
| D#n | 0 through 9 | Decimal Value |
| H#n | 0 through 9 or A through F | Hexadecimal Value |

$ Use the current program counter as the value (relative addressing)

## 3.13 Constant Lengths

The length of the constant may be given explicitly. For example:

B: EQUΔ 4D#8

Where

4 is the explicit length and the value is 1000

If an explicit length is not given, the constant is given an implicit length. The implicit length is determined by the designator used.

**Table 3-1  Implicit Length Attributes of Constants**

| Expression | Implicit Length | Binary Value | Description |
|---|---|---|---|
| B: EQUΔ B#1000 | 4 | 1000 | Each binary digit yields an implicit length of 1 bit per digit. |
| B: EQUΔ Q#10 | 6 | 001 000 | Each octal digit yields an implicit length of 3 bits per digit. |
| B: EQUΔ H#10 | 8 | 0001 0000 | Each hexadecimal digit yields an implicit length of 4 bits per digit. |
| B: EQUΔ 12 | 4 | 1100 | The 12 is assumed to be decimal, and the implicit length is counted from the rightmost bit through the leftmost 1. |
| B: EQUΔ 3 | 2 | 11 | Same as above. Implicit length 2. |
| B: EQUΔ 4 | 3 | 100 | Same as above. Implicit length 3. |

## *3.14 Constant Modifiers*

Constants may have modifiers following their given value (n). Modifiers are inversion (*), negation (-), truncation (:) and right justification (%). They must appear after the constant digits where they may be in any order but they will be processed in the following order:

| Modifier | Description |
|---|---|
| * - | Inversion or negation |
| % | Right Justification |
| : | Left Truncation |

Thus, a negative constant is indicated by n- which produces the two's complement of n.

A constant may not be modified by both inversion and negation.

If a constant including modifiers is given as a VFS, any attributes (permanent modifiers) given for that field in the Definition File will also modify the value of the constant given.

If, for example, the Definition file contains:

```
A: DEF 5X, 3V*B#, 2X, 5V%H#, B#10101
           Field #1    Field #2
```

And the Assembly File is written:

```
TEST: AΔ001, 9
```

The binary value 001 is inverted and substituted for field #1, while the 9 (Hex) is equated to binary 1001 and right justified for field #2 resulting the microinstruction:

```
XXXXX   110   XX   01001   10101
```

If the assembly File statement is written

```
TEST2: AΔ 001*, 3*
```

The binary value 001 is inverted by the current * attribute, then inverted again by the * attribute in the Definition File for Field #1. Hex 3 (binary 0011) is inverted to 1100 and right justified in Field #2.

The complete microinstruction is:

```
XXXXX   001   XX   01100   10101
```

## 3.15 Variable Field Substitution (VFS)

When the formats are defined in the Definition File, some of the fields may be designated as variable fields. If these fields are not given a default value during their definition or if one wishes to override the default value of one or more fields, a substitution must be made for these fields in the Assembly File source statements.

These substitutions must obey the following rules:

### 3.15.1 Required Substitutions

If the variable fields are not given default values in the Definition file, values for these fields must be provided in the assembly File source statements. If omitted, an error message will be provided, and processing of that statement ends.

### 3.15.2 Substitution Separators

Each VFS (whether required or optional) represents a single field and must be separated from other VFSs by a comma. Trailing commas may be omitted. Note that the assembler uses the commas to indicate which fields are to be given substitute values (i.e., VFSs are positional and position is determined by the number of commas), so leading and intermediate commas must be given.

For example, if the Definition File contains:

$$\text{A: DEF 5X, }\underbrace{\underline{\text{3V*B\#110}}}_{\text{Field \#1}}\text{, 2X, }\underbrace{\underline{\text{5V\%H\#}}}_{\text{Field \#2}}\text{, B\#10101}$$

If the assembly File statement is written 3V*B#110

```
TEST3: AΔ , 4
```

Field #1 will assume the default value 001 (from 3V*B#110) while field #2 will be equated to 0100 and right justified in the 5-bit field so that field is 00100..

The complete microinstruction is:

```
XXXXX   001   XX   00100   10101
```

If the comma were omitted and

```
TEST4: AΔ  4      ; missing comma means wrong value to field #1 and no value for field #2
```

Were written, the assembler would try to use 4 as the VFS for field #1. Two errors are present. The 4 is not a binary number as required for field #1, and no value is indicated for field #2. Field #2 had no explicit default value, and no VFS is given which is an error. The indicated error would be “illegal character”, since the 4 is assumed to go with field #1 which requires binary digits.

If, however, the user wished to input field #1 as an octal 4 and field #2 as zero, the statement could be written:

```
TEST5: AΔ  Q#4, 0
```

Which yields the microinstruction

$$\text{XXXXX}\quad \underbrace{\underline{\text{011}}}_{\substack{\text{Octal 4}\\ \text{Inverted}}}\quad \text{XX}\quad \underbrace{\underline{\text{00000}}}_{\substack{\text{Hex 0}\\ \text{right-justified}}}\quad \text{10101}$$

In short, when forming the microword definition, if a leading or intermediate variable field is to assume a default value but a trailing field requires a VFS, each field to be skipped must be represented by a comma.

This is best explained by an example. Assume a format ADE with three variable fields, each having a default value of zero specified in the Definition File:

ADE:  DEFΔ  3VB#000, 3VB#000, 3VB#000

The following example illustrates fields which assume their default values and fields which are given override or substitute values.

| Instruction | Resultant Microword Definition | Meaning |
|---|---|---|
| TEST6: ADEΔ ,,010<br>Or<br>TEST7: ADEΔ ,, Q#2 | 000 000 010<br><br>000 000 010 | Fields 1 and 2 assume their default values, field 3 contains 010. |
| TEST6: ADEΔ  Q#4,, B#101 | 100 000 101 | Fields 2 assumes its default value, field 1 is 100, field 3 contains 101. |
| TEST6: ADEΔ B#011 | 011 000 000 | Fields 2 and 3 assume their default values, field 1 contains 011. |

If the variable field substitutions contain modifiers, using the Definition File statement:

ADE:  DEFΔ  3VB#000, 3VB#000, 3VB#000

The Assembly File statements for the previous example could be written as:

| Instruction | Resultant Microword Definition | Meaning |
|---|---|---|
| TEST10: ADEΔ ,,101* | 000 000 010 | Fields 1 and 2 assume their default values, field 3 contains 101 inverted. |
| TEST11: ADEΔ  H#4: | 100 000 101 | Field 1 is Hex 4 (binary 0100) truncated to 100. Fields 2 and 3 assume their default values. |

The variable fields may contain attributes in the Definition file such as:

ADE:  DEFΔ  3V:H#0, 3V*B#000, 3V%B#000

The Assembly File statements written below now generate different data than in the previous example:

| Instruction | Resultant Microword Definition | Meaning |
|---|---|---|
| TEST12: ADEΔ ,,01* | 000 111 010 | Fields 1 assumes its default value 000. Field 2 assumes the default value 111. (000 inverted.) Field 3 is inverted to 10 then right-justified to be 010. |
| TEST13: ADEΔ 9, Q#3*, 1 | 001 011 001 | Field 1 is binary 1001 truncated to 001. Field 2 is octal 3 inverted to 100, then inverted by field #2 attribute (*) to 011. Field 3 is binary 1 right-justified to 001. |

### 3.15.3  Fitting Variable Substitutes to Variable Fields

Any value given as a variable field substitution (VFS) must contain exactly the number of bits specified (in the Definition Field) for the total length of the variable field unless the modifiers % (right-justify), : (truncation), or $ (paged addressing) are given.

These modifiers may be supplied as attributes with the original field definition (Definition file) or they may be supplied with the filed substitution value in the Assembly File.

### 3.15.4 Paged and Relative Addressing

$ is used in two ways in the Assembly file:

a) To indicate that the current value of the program counter is the value to be substituted into this field. This is called *relative addressing*.

b) As an attribute to indicate that the value substituted for this field must be on the same “memory page” as the microword into which it is substituted. This is called *paged addressing*.

For relative addressing, the $ alone or as part of the expression is used as a VFS.

For paged addressing, the $ may be given as an attribute of this variable field in the Definition File, or the $ may immediately follow the VFS in the Assembly File source statement.

For example, if the Definition File contains

```
JSR: DEF 8X, 8V$, H#, 27, 12V

JSB: DEF 8V%D#, 8X, 8Q#013:, 12X
```

The Assembly File can be written:

```
Line #
  1        JSR       BEGIN, H#0BC
  2        JSB         MULT$ + 5
  3        JSR       MULT, BEGIN$
  4        JSB           H#37
  5        JKSB          $ + 5
            •
            •
            •
          BEGIN:          ADD
            •
            •
            •
          MULT:           MPY
```

Line 1-3 are example of $ used for paged addressing. In Line 1, the value of the program counter where BEGIN : appears is substituted into the first variable field of the format JSR. This value is left truncated if necessary to fit into this 8-bit field, and any truncated left bits must be identical to the corresponding bits of the program counter associated with Line 1.

The same type of substitution, truncation, etc. occurs for Lines 2 and 3.

Note that:

- The JSB on line 2 needs a $ after MULT if paged addressing is desired since no $ was given with that variable field in the Definition File.
- The JSR on Line 1 needs no $ with the BEGIN since that variable field does contain a $ in the Definition File.
- The JSR on Line 3 requires a $ after BEGIN since the second variable field did not contain a $ in the definition file.
- On Line 2, a label with a $ may be part of an expression.

Line 5 is an example of relative addressing. The current value of the program counter plus 5 will be substituted for the variable field.

Note that:

- There is no connection between the $ used for paged addressing—and the $ used as a variable field substitute to indicate use of the current program counter value (relative addressing).

## 3.16 Hexadecimal Attribute

The designator H#, if given with a variable field in the Definition File, is a permanent attribute but may need to be repeated in the Assembly File. This is necessary since the program cannot distinguish a hexadecimal value, which begins with an A through F, from a label or format name.

IF H# is a permanent attribute for a variable field, the digits 39 used as a VFS would be interpreted as hexadecimal (0011 1000) when encountered in the Assembly File. However, whenever a value is encountered in the Assembly file where the variable definition includes the implicit hexadecimal type (e.g., 8VH#) and the digit is one of the letters “A” through “F”, the value must be preceded by the H# designator (e.g., H#BAD) to distinguish it from a symbol of the same name (e.g., VAD: EQU 123).

## 3.17 Expressions

Expression may be used when the programmer wishes to have a value calculated for a constant, address, or VFS. An expression assumes the form:

---

form:

---

symbol operator symbol operator . . .

---

For example:

    SBBΔ 1 + 2

Is the same as SBBΔ 3 (and 2 are expression symbols, + is an expression operator). The expression

    JMPΔ $ - 5

Yields the current value of the program counter minus 5 as the VFS for the first variable field in the format name JMP. ($ and 5 are expression symbols, - is an expression operator). The expression

    EIGHT: EQUΔ  2*2*2

Means EIGHT = 8 (2’s are the expression symbols, *’s are an expression operators).

All arithmetic done in expressions is integer. Remainders are discarded.

The result of any expression must be a positive constant.

### 3.17.1 Expression Operators

Operators permitted in expressions are:

| Operator | Description |
|---|---|
| + | Add the value of the left symbol (the symbol on the left of the +) to the value of the right symbol (the symbol on the right of the +) |
| - | Subtract the value of the left symbol (the symbol on the left of the -) to the value of the right symbol (the symbol on the right of the -) |
| * | Multiply the left symbol by the right symbol |
| / | Divide the symbol on the left (dividend) by the symbol on the right (divisor) |

### 3.17.2 Order of Expression Evaluation

Expressions are always evaluated from left to right. Thus,

```
A-B*2
```

Adds A to the negative of B and multiplies this value by 2.

An expression is terminated by a comma or the end of the line except when used as a field in FF where it is enclosed by parenthesis. To continue an expression on the next line the first nonblank character must be a slash (/). A continuation involving a division would thus require a double forward slash (//).

## *3.18 Assembler Output*

The Assembly Phase output consists of:

- An image of the associated assembly file input statement and error messages.
- Optional listing of the symbol table.
- Optional listing for entry points.
- A choice of one of four types of printed listings.

| Type | Description |
|---|---|
| I | Interleaved format (INTER). One line of source code is printed with the corresponding line of object code printed directly below it. |
| II | Source-only format (SRCONLY). Only the Assembly File source statements are printed. |
| III | Object code only format (OBJONLY). Only the Assembly Phase binary code is printed. |
| IV | Block format (BLOCK). All lines of source code are printed followed by all lines of the object code. |

Each of these listings contains the location (program counter) associated with each line of the object code.

A final option is to output the binary code directly to disk for use as input to the post-processing phase. (Disk output is independent of the listing option chosen.) The object code on the disk may then be used, for example, as input to the post-processing phase, which might punch a paper tape in a format suitable for burning PROMs. [SIC this is 1977 remember. Today, we would input to digital DVD/CD reader on a PC set up to burn a PROM/EPROM these days or more than likely over an internet or wireless connection which feeds directly to the unit chosen to program or “burn” the microcode storage devices.]

## *3.19 Assembler Symbol Table*

The symbol table contains a list of all the symbols (constant names) defined by the EQUs and all labels used on Assembly File source statements. The symbol table also includes all the constant names and their associated values defined using EQUs in the Definition File.

For each symbol, the table lists the label and the program counter value of the statement where the label is defined or if the symbol is a constant name (defined by EQU) it is followed by the value of the constant.

A symbol table is useful when errors occur due to misspelling or the omission of the colon after a label.

A sample symbol table is:

```
SYMBOLS
        A       0001
        S       0023
        X       0000
```

Printing of the symbol table is optional and is described in the SYMBOL and NOSYMBOL section of Table 4-1.

## 3.20 Assembler Entry Point Table

The entry point table contains a list of all the entry point symbols (labels followed by ::) and their associated program counters. These values are useful for mapping PROMs.

Printing of the entry point table is optional and is described in the MAP and NO MAP section of Table 4-1.

## 3.21 Assembly File – Reserved Words

The following are reserved words used by the assembler program during the Assembly Phase. These words may not be used as labels in the Assembly File statements.

ALIGN
EJECT
END
FF
LIST
NOLIST
ORG
RES
SPACE
TITLE
Format names or constant names from the Definition File.

## 3.22 Assembly Phase Error Messages and Interpretations

The possible Assembly Phase error messages and their meanings are listed below:

ERROR 1 ILLEGAL CHARACTER

This message will most commonly occur when a comma or blank is missing, but may occur when the wrong character is struck on the keyboard or the wrong designator was used. (B#3, Q#9 are inconsistent.)

ERROR 2 UNDEFINED SYMBOL

This error will occur when a constant name or label is misspelled or not defined. It may also occur if the semicolon is omitted before a comment or if the blank is missing after RES, ORG, FF, EQU, TITLE, ALIGN or SPACE. It will occur when the VFS for a hexadecimal field begins with A through F and the H# is not given before the letter.

ERROR 3 UNDEFINED FORMAT

The format name given is misspelled or was not defined in the Definition Phase or the required blank was not supplied after the format name.

ERROR 5 DUPLICATE LABEL

This label has been used more than once as a constant name or a label. If the label is more than 8 characters, the first 8 must be unique.

ERROR 7 FORMAT FIELD OVERFLOW

In this FF statement more than 30 fields have been given. Since the maximum number of fields permitted per FF is 30, some of the fields must be combined.

ERROR 9 UNDEFINED DIRECTIVE

This is usually caused by a missing colon, blank, or misspelling. The assembler was searching for ORG, RES, SPACE, etc., and this symbol does not match the list of such 'directives'.

ERROR 10 ILLEGAL MICROWORD LENGTH

The total bits given for this microword do not match the word length given in the Definition Phase.

ERROR 11 ILLEGAL FIELD LENGTH

No field, except a "don't care" field, may be more than 16 bits in length. The value calculated for this field cannot be represented in 16 bits.

ERROR 12 DON'T CARE FIELD TOO LONG.

A "don't care" field has been defined, with a length larger than the length of the microword. Look for missing commas or designators.

ERROR 13 ARITHMETIC OPERATION ON FIXED FIELD.

If a field is defined as a variable field in the Definition File, an expression cannot be used as a VFS in the Assembly File unless the field contained the % attribute in its definition.

ERROR 14 ATTRIBUTE ERROR

This error occurs when * and − are both associated with a single field.

ERROR 16 MISSING END STATEMENT

The Definition or the Assembly File is missing the END statement.

ERROR 17 ILLEGAL SYMBOL

This error occurs when a 6 is typed instead of an &, a comma is missing, or some character not permitted (e.g. ! ' etc.) has been input inadvertently.

ERROR 18 OVERLAY ERROR

This message is given when two formats are overlayed and both of them contain constants for the same bit position. If the assembler is run using each of the formats in the overlay statement as a separate format, and the output is printed in block form, the erroneous bits are easily detected.

For example if the Definition File statements are:

```
A: DEF 4X, B#1011
B: DEF B#01111, 3X
```

and the Assembly File statement is

```
A & B
```

the overlay error message occurs.

Rerun the Assembly File with source statements given as

```
A
B
```

and block output requested which generates

$$\begin{array}{lcl} \text{XXXX} & \boxed{1} & 011 \\ 0111 & \boxed{1} & \text{XXX} \end{array}$$

It can easily be seen that bits $\begin{array}{c}\boxed{1}\\ \boxed{1}\end{array}$ are causing the overlay error. The improper DEF can then be corrected and A & B used in the Assembly File statement.

ERROR 19 NO DEFAULT VALUE

A format name was defined with a variable field in the Definition File. Since no default value was given in the definition, a variable field substitute **must** be supplied for this field when the format name is used in the Assembly File. Check for missing commas.

## ERROR 20 FIELD LENGTH CONFLICT

The value specified as a VFS with a format name or the digits given for a field in an FF statement do not match the explicit length for this field.

## ERROR 21 $ SPECIFIED FOR NON-ADDRESS FIELD

In order to use the value of the program counter (indicated with a $) as a VFS, that field must contain the % attribute.

## ERROR 23 MISSING DESIGNATOR

If a comma is missing in the FF statement, the assembler may encounter a digit or designator as part of a "don't care" field.

For example, erroneous statements are:

```
FFΔ 12X 3B#101
FFΔ 12X Q#5
```

## ERROR 24 SPACE DIRECTIVE ERROR

The value input following SPACE is interpreted as less than zero or greater than the number of lines given per page.

## ERROR 25 ORG SET TO LESS THAN CURRENT PC

When ORG is encountered, the value given is compared with the current program (location) counter. If ORG is less than the program counter, the value given with ORG is ignored.

## ERROR 26 NO FORMAT NAME AFTER &

When a line ends with an & and no continuation (/) is given at the beginning of the next line, this error is generated. A format name is missing after the &, or a / is missing on the continuation line.

## ERROR 28 ADDRESS NOT IN CURRENT PAGE

When the user gives a label or a label$ as a VFS or has defined his variable field with the $ attribute, this message will be generated if the left bits to be truncated do not match the corresponding bits of the current program counter.

## ERROR 29 LENGTH REQUIRED FOR $ MODIFIER

Paged addressing (use of the $ as a modifier) requires the field length before the symbol. Thus, 6SYMBOL$ is correct, but SYMBOL$ is incorrect.

## ERROR 30 ILLEGAL FIELD LENGTH IN FF STMT.

A field is greater than 16 bits in a FF statement. Only "don't care" fields may be larger than 16 bits.

## ERROR 32 NO EXPLICIT LENGTH BEFORE (

An expression in a FF statement must be enclosed in (). The explicit field length must precede the (.

## 3.23 Error Messages which Halt Execution

Error messages with n ⩾ 100 cause execution to stop. They are listed below:

**ERROR 100   COMMAND OPTION SYNTAX ERROR**

The input command contains an error. Check for correct spelling of filenames, spaces between options, complete sets of parenthesis and correct drive specification with filenames.

**ERROR 101   DEF TABLE OVERFLOW**

**ERROR 102   SUB TABLE OVERFLOW**

**ERROR 103   EQU TABLE OVERFLOW**

**ERROR 106   FIELD TABLE OVERFLOW**

Errors 101, 102, 103 and 106 occur when the amount of memory available has been exceeded.

**ERROR 104   INCORRECT OR MISSING WORD SIZE**

Either the WORD n command is not given as the first command (or the first command after TITLE) or the value given for n is < 1 or > 128.

**ERROR 105   UNEXPECTED END OF FILE**

The user has given an incorrect file name or the source file is not correct. AMDASM has encountered an end of file when it was still expecting data.

# 4 Chapter IV

## 4.1 AMDASM/80 Execution (The Original System)

After the user has created the Definition File and the Assembly File using the MDS text editor (or any text editor), then AMDASM/80 can be executed. After the ISIS© operating system has issued a user prompt (i.e., a "–" character) the microassembler is executed by entering the command: [If you are running your own meta-assembler – follow the instructions for running it.]

—AMDASMΔ PHASEn(filename) Δ {options}

PHASE1 (filename) specifies the execution of the Definition Phase using (filename) for the definition source file.

PHASE2 (filename) specifies the execution of the Assembly Phase using (filename) for the assembly source file.

PHASE1 (filename) and PHASE2 (filename) specifies execution of both the Definition and Assembly Phases.

Thus

—AMDASMΔ PHASE1(:F1:DEFN) specifies execution of only the Definition Phase using the file (on drive 1) called DEFN.

or

—AMDASMΔ PHASE1(:F1:DEFN) PHASE2(:F1:ASMSRC) specifies execution of the Definition and Assembly Phases using the files (on drive 1) DEFN as the definition source file and ASMSRC as the assembly source file.

Either PHASE 1 (or P1) or PHASE2 (or P2) or both must be specified following ASDASMΔ. The user then enters the desired options. Options and their default values are shown in Table 4-1 (next page). The full option may be typed (OBJECT) but only the alternate option (O) needs to be typed.

In the option table (Table 4-1), filename must be an ISIS filename of the form:

:device:name.ext

where:

| | |
|---|---|
| :device: | is optional and is :F0:, :F!:, :F2:, or :F3: to indicate the drive on which the diskette is mounted. If omitted, :F0: is assumed. |
| name | Is from 1 to 6 uppercase letters or digits and is required. |
| .ext | Is a period followed by 1 to 3 uppercase letters or digits and is optional |

Options need to be separated by at least one blank character for other options in the command. If an option ends with ), the blank space is not needed.

Whenever a user does not specify an option in the execution command, AMDASM will use the default given in the preceding table.

The command language for executing AMDASM is best illustrated with examples.

–AMDASMΔP1 (DEFN.SRC)P2(MUCODE.SRC)

specifies execution of both PHASE 1 and PHASE 2 using DEFN.SRC as the input file for PHASE 1 and MUCODE.SRC for Phase 2. Defaults are selected for all other options.

–AMDASMΔPHASE1 (DEFN.SRC) DEF (AM9080.DEF)

specifies execution of PHASE 1 with DEFN.SRC as the input source file and AM9080.DEF as the definition table output file.

```
–AMDASMΔPHASE2 (MUCODE.SRC) DEF(AM9080.DEF) INTERΔNOSYMBOL
```

specifies execution of PHASE 2 with MUCODE.SRC as the input source file and AM9080.DEF as the definition table input file, interleaved listing format, a list of entry point symbols, and no symbol table listing.

The interleaved format prints a line of source code followed by a line of object code.

The block format prints all lines of source code, then all lines of object code.

The source-only format prints only the source code.

The object-only format prints only the object code.

## Table 4-1 AMDASM/80 Options

| OPTION | ALTERNATE OPTION | DEFAULT | MEANING |
|---|---|---|---|
| DEF (filename) | D | (AMDASM.DEF) | Specifies the name of the file where output of the Definition Phase is to be stored. When only PHASE2 is executed, this specifies the input file which contains the processed definitions. If no DEF (filename) is given, the default name AMDASM.DEF will be used. |
| LIST (filename) | L | (AMDASM.LST) | Specifies where the Definition and Assembly output is to go. (:LP:) for the filename causes the output to be listed on the line printer. If no LIST (filename) is given, the output goes to the file with the default name (AMDASM.LST) |
| NOLIST | NL | | Suppresses listing of assembly source code. |
| OBJECT (filename) | O | (AMDASM.OBJ) | Specifies that the microcode (object code) is to be output on a file with the name (filename). If not given, the microcode is placed on a file with the default name AMDASM.OBJ. |
| NOOBJECT | NO | | Suppresses placement of the microcode onto a file. If block format printing is requested, the object code printing is also suppressed. |
| INTER | IL | BLOCK | Specifies inter-leaved listing format. |
| BLOCK | BL | | Specifies blocked listing format |
| SRCONLY | SO | | Specifies source-only listing format |
| OBJONLY | OB | | Specifies object-only listing format |
| WIDTH (n) | W | n=72 | Specifies width n, (a decimal number) in characters of listing device. |
| LINES (n) | LN | n=66 | Specifies number n, (a decimal number) of lines per page. If not specified, default is 66 lines (11 inches). |
| MAP | M | MAP | Specifies listing of entry point symbols (i.e., label symbols designated as entry points by double colons "::") |
| NOMAP | NM | | Suppresses listing of entry point symbols If not specified, defaults to MAP |
| HEX | H | HEX | Specifies listing of location counter in hexadecimal format |
| OCTAL | Q | | Specifies listing of location counter in octal format. If not specified defaults to HEX. |
| SYMBOL | S | SYMBOL | Specifies listing of constant names and labels and their associated values. |
| NOSYMBOL | NS | | Supresses listing of Symbol table. If not specified, defaults to SYMBOL |

# 5 Chapter V

## 5.1 *Sample of AMDASM/80 Processing*

The capabilities of AMDASM/80 can be demonstrated by microprogramming one of the exercises from the Am2900 Learning and Evaluation kit. This kit provides a simple but complete example of a microprogrammed system.

The architecture of the kit is shown in Figure 5-1. The dashed lines outline the two LSI components, the Am2909 microprogram sequencer, and the Am2901 four-bit bit-slice microprocessor. Each microinstruction in the microprogram memory consists of 32 bits divided into fields to control the sequencer, branch address, shift multiplexers, and all the inputs to the Am2901. The fields and their functions are defined in Figure 5-2.

The first step in using AMDASM/80 is the creation of a set of definitions, which reflect the hardware on which the microprogram will run. The statements in Figure 5-3 completely define, mnemonically, the fields in the kit. That is, they implement exactly the fields and their functions for the microprocessor architecture defined in Figure 5-1, and so may be used in writing all microprograms that aree to operate in this architecture. Figure 5-4 shows a flowchart of the program to be written. Figure 5-5 is the AMDASM output in Block format.

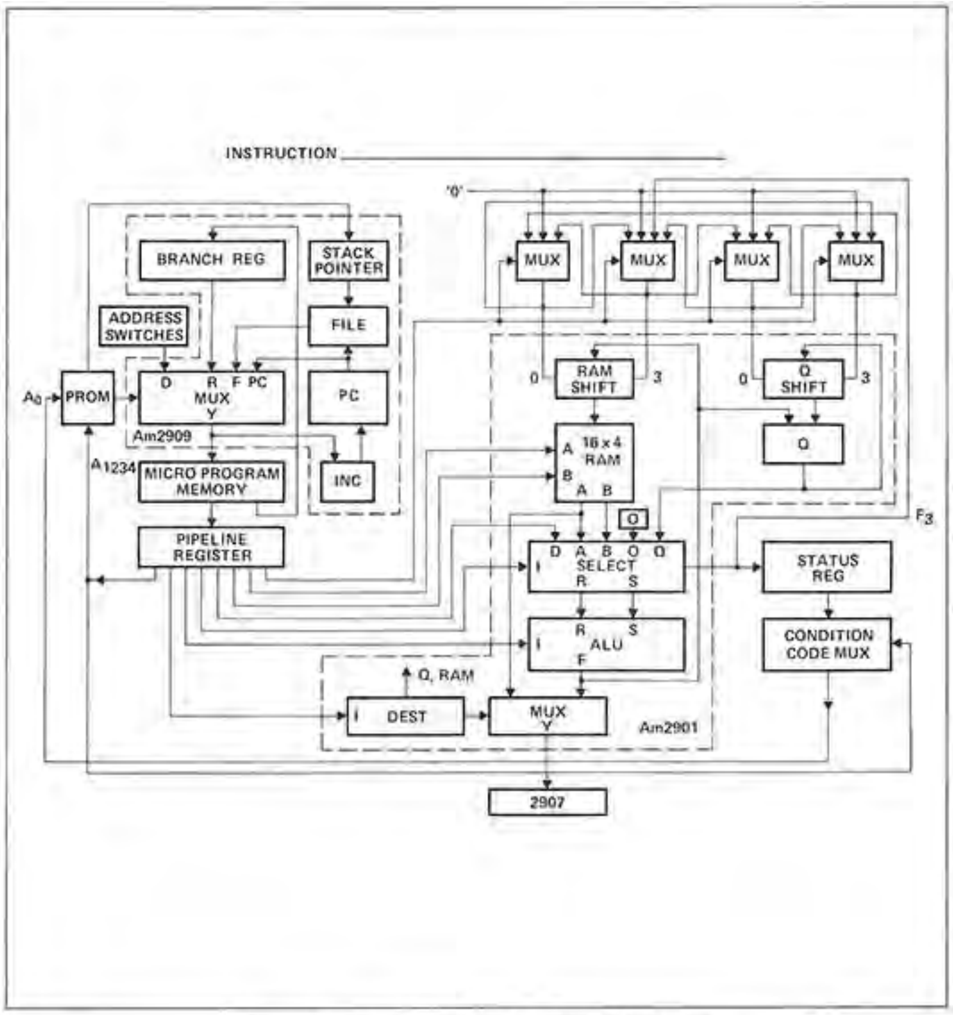
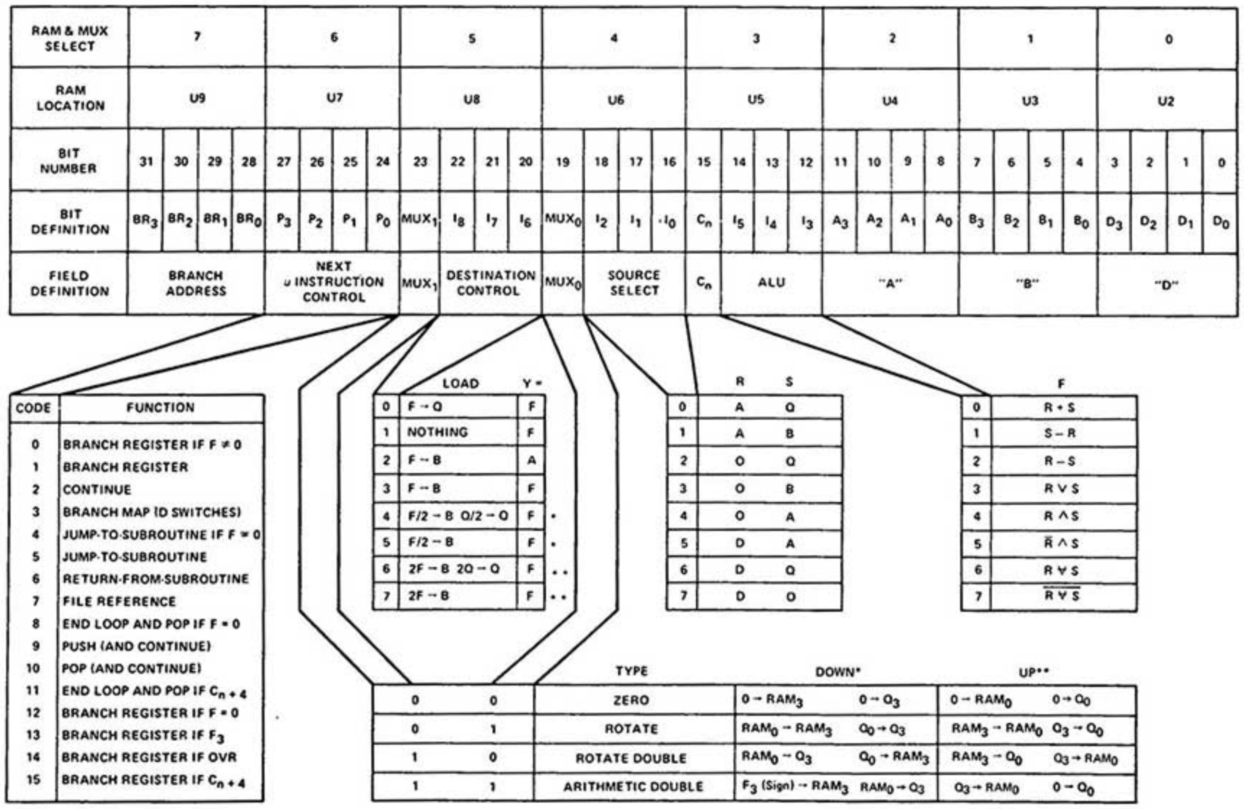


**Figure 5-1 Am2900 Learning and Evaluation Kit Architecture**

**Figure 5-2 Example of Fields and Functions**



| RAM & MUX SELECT | 7 | | | | 6 | | | | 5 | | | | 4 | | | | 3 | | | | 2 | | | | 1 | | | | 0 | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| RAM LOCATION | U9 | | | | U7 | | | | U8 | | | | U6 | | | | U5 | | | | U4 | | | | U3 | | | | U2 | | | |
| BIT NUMBER | 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| BIT DEFINITION | $BR_3$ | $BR_2$ | $BR_1$ | $BR_0$ | $P_3$ | $P_2$ | $P_1$ | $P_0$ | $MUX_1$ | $I_8$ | $I_7$ | $I_6$ | $MUX_0$ | $I_2$ | $I_1$ | $I_0$ | $C_n$ | $I_5$ | $I_4$ | $I_3$ | $A_3$ | $A_2$ | $A_1$ | $A_0$ | $B_3$ | $B_2$ | $B_1$ | $B_0$ | $D_3$ | $D_2$ | $D_1$ | $D_0$ |
| FIELD DEFINITION | BRANCH ADDRESS | | | | NEXT μ INSTRUCTION CONTROL | | | | $MUX_1$ | DESTINATION CONTROL | | | $MUX_0$ | SOURCE SELECT | | | $C_n$ | ALU | | | "A" | | | | "B" | | | | "D" | | | |

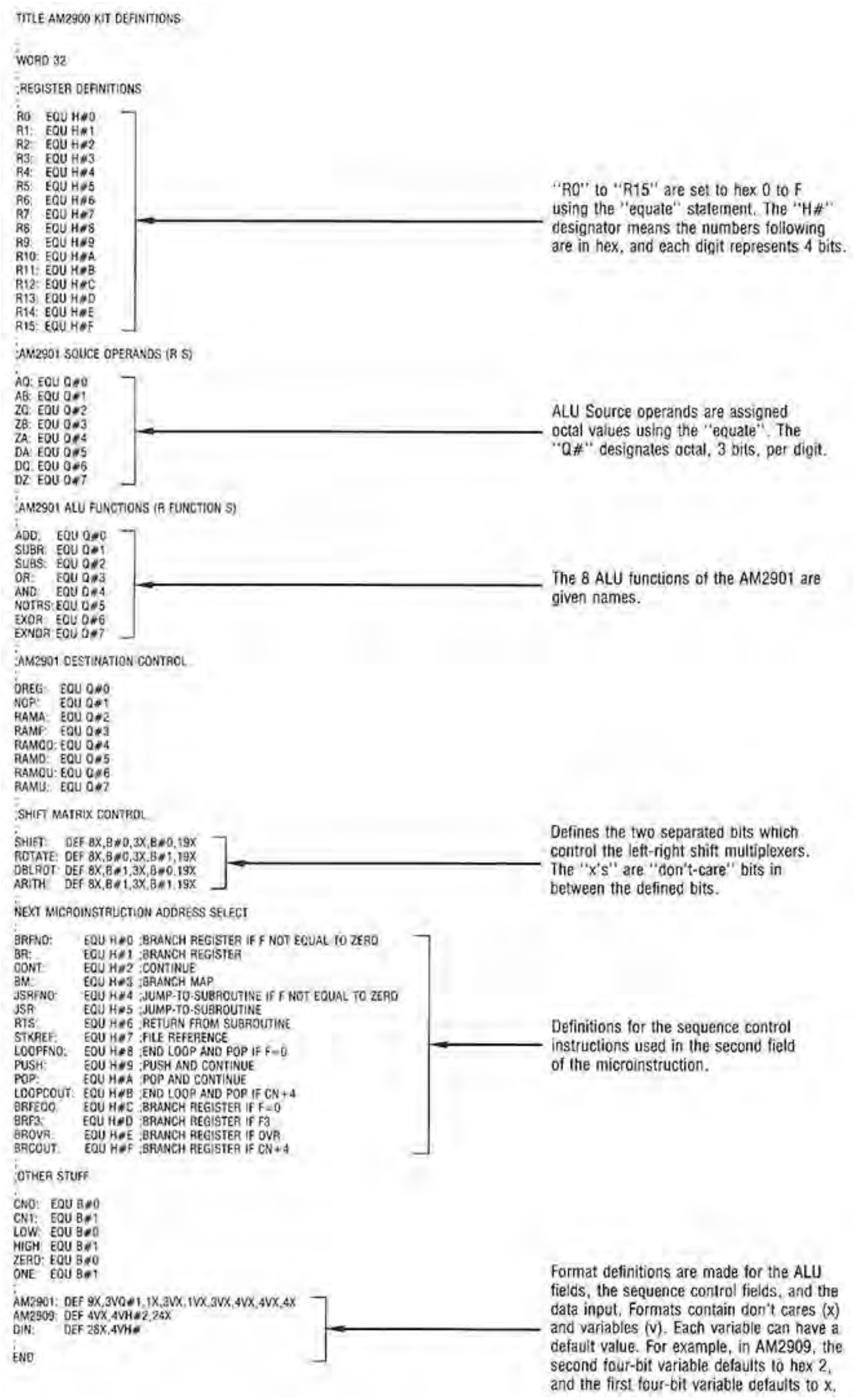
| CODE | FUNCTION |
|---|---|
| 0 | BRANCH REGISTER IF F ≠ 0 |
| 1 | BRANCH REGISTER |
| 2 | CONTINUE |
| 3 | BRANCH MAP (D SWITCHES) |
| 4 | JUMP-TO-SUBROUTINE IF F = 0 |
| 5 | JUMP-TO-SUBROUTINE |
| 6 | RETURN-FROM-SUBROUTINE |
| 7 | FILE REFERENCE |
| 8 | END LOOP AND POP IF F = 0 |
| 9 | PUSH (AND CONTINUE) |
| 10 | POP (AND CONTINUE) |
| 11 | END LOOP AND POP IF $C_{n+4}$ |
| 12 | BRANCH REGISTER IF F = 0 |
| 13 | BRANCH REGISTER IF $F_3$ |
| 14 | BRANCH REGISTER IF OVR |
| 15 | BRANCH REGISTER IF $C_{n+4}$ |

| | LOAD | Y = | |
|---|---|---|---|
| 0 | $F \rightarrow Q$ | F | |
| 1 | NOTHING | F | |
| 2 | $F \rightarrow B$ | A | |
| 3 | $F \rightarrow B$ | F | |
| 4 | $F/2 \rightarrow B$ $Q/2 \rightarrow Q$ | F | * |
| 5 | $F/2 \rightarrow B$ | F | * |
| 6 | $2F \rightarrow B$ $2Q \rightarrow Q$ | F | ** |
| 7 | $2F \rightarrow B$ | F | ** |

| | R | S |
|---|---|---|
| 0 | A | Q |
| 1 | A | B |
| 2 | O | Q |
| 3 | O | B |
| 4 | O | A |
| 5 | D | A |
| 6 | D | Q |
| 7 | D | O |

| | F |
|---|---|
| 0 | R + S |
| 1 | S − R |
| 2 | R − S |
| 3 | $R \vee S$ |
| 4 | $R \wedge S$ |
| 5 | $\bar{R} \wedge S$ |
| 6 | $R \veebar S$ |
| 7 | $\overline{R \veebar S}$ |

| | | TYPE | DOWN* | | UP** | |
|---|---|---|---|---|---|---|
| 0 | 0 | ZERO | $0 \rightarrow RAM_3$ | $0 \rightarrow Q_3$ | $0 \rightarrow RAM_0$ | $0 \rightarrow Q_0$ |
| 0 | 1 | ROTATE | $RAM_0 \rightarrow RAM_3$ | $Q_0 \rightarrow Q_3$ | $RAM_3 \rightarrow RAM_0$ | $Q_3 \rightarrow Q_0$ |
| 1 | 0 | ROTATE DOUBLE | $RAM_0 \rightarrow Q_3$ | $Q_0 \rightarrow RAM_3$ | $RAM_3 \rightarrow Q_0$ | $Q_3 \rightarrow RAM_0$ |
| 1 | 1 | ARITHMETIC DOUBLE | $F_3$ (Sign) $\rightarrow RAM_3$ | $RAM_0 \rightarrow Q_3$ | $Q_3 \rightarrow RAM_0$ | $0 \rightarrow Q_0$ |



```
TITLE AM2900 KIT DEFINITIONS
;
;
WORD 32
;
;
;REGISTER DEFINITIONS
;
R0:   EQU H#0
R1:   EQU H#1
R2:   EQU H#2
R3:   EQU H#3
R4:   EQU H#4
R5:   EQU H#5
R6:   EQU H#6
R7:   EQU H#7
R8:   EQU H#8
R9:   EQU H#9
R10:  EQU H#A
R11:  EQU H#B
R12:  EQU H#C
R13:  EQU H#D
R14:  EQU H#E
R15:  EQU H#F
;
;AM2901 SOUCE OPERANDS (R S)
;
AQ: EQU Q#0
AB: EQU Q#1
ZQ: EQU Q#2
ZB: EQU Q#3
ZA: EQU Q#4
DA: EQU Q#5
DQ: EQU Q#6
DZ: EQU Q#7
;
;AM2901 ALU FUNCTIONS (R FUNCTION S)
;
ADD:    EQU Q#0
SUBR:   EQU Q#1
SUBS:   EQU Q#2
OR:     EQU Q#3
AND:    EQU Q#4
NOTRS:  EQU Q#5
EXOR:   EQU Q#6
EXNOR:  EQU Q#7
;
;AM2901 DESTINATION CONTROL
;
QREG:   EQU Q#0
NOP:    EQU Q#1
RAMA:   EQU Q#2
RAMF:   EQU Q#3
RAMQD:  EQU Q#4
RAMD:   EQU Q#5
RAMQU:  EQU Q#6
RAMU:   EQU Q#7
;
;SHIFT MATRIX CONTROL
;
SHIFT:   DEF 8X,B#0,3X,B#0,19X
ROTATE:  DEF 8X,B#0,3X,B#1,19X
DBLROT:  DEF 8X,B#1,3X,B#0,19X
ARITH:   DEF 8X,B#1,3X,B#1,19X
;
NEXT MICROINSTRUCTION ADDRESS SELECT
;
BRFNO:      EQU H#0 ;BRANCH REGISTER IF F NOT EQUAL TO ZERO
BR:         EQU H#1 ;BRANCH REGISTER
CONT:       EQU H#2 ;CONTINUE
BM:         EQU H#3 ;BRANCH MAP
JSRFNO:     EQU H#4 ;JUMP-TO-SUBROUTINE IF F NOT EQUAL TO ZERO
JSR:        EQU H#5 ;JUMP-TO-SUBROUTINE
RTS:        EQU H#6 ;RETURN FROM SUBROUTINE
STKREF:     EQU H#7 ;FILE REFERENCE
LOOPFNO:    EQU H#8 ;END LOOP AND POP IF F=0
PUSH:       EQU H#9 ;PUSH AND CONTINUE
POP:        EQU H#A ;POP AND CONTINUE
LOOPCOUT:   EQU H#B ;END LOOP AND POP IF CN+4
BRFEQO:     EQU H#C ;BRANCH REGISTER IF F=0
BRF3:       EQU H#D ;BRANCH REGISTER IF F3
BROVR:      EQU H#E ;BRANCH REGISTER IF OVR
BRCOUT:     EQU H#F ;BRANCH REGISTER IF CN+4
;
;OTHER STUFF
;
CN0:   EQU B#0
CN1:   EQU B#1
LOW:   EQU B#0
HIGH:  EQU B#1
ZERO:  EQU B#0
ONE:   EQU B#1
;
AM2901:  DEF 9X,3VQ#1,1X,3VX,1VX,3VX,4VX,4VX,4X
AM2909:  DEF 4VX,4VH#2,24X
DIN:     DEF 28X,4VH#
;
END
```

"R0" to "R15" are set to hex 0 to F using the "equate" statement. The "H#" designator means the numbers following are in hex, and each digit represents 4 bits.

ALU Source operands are assigned octal values using the "equate". The "Q#" designates octal, 3 bits, per digit.

The 8 ALU functions of the AM2901 are given names.

Defines the two separated bits which control the left-right shift multiplexers. The "x's" are "don't-care" bits in between the defined bits.

Definitions for the sequence control instructions used in the second field of the microinstruction.

Format definitions are made for the ALU fields, the sequence control fields, and the data input. Formats contain don't cares (x) and variables (v). Each variable can have a default value. For example, in AM2909, the second four-bit variable defaults to hex 2, and the first four-bit variable defaults to x.

**Figure 5-3 Definition FIle**

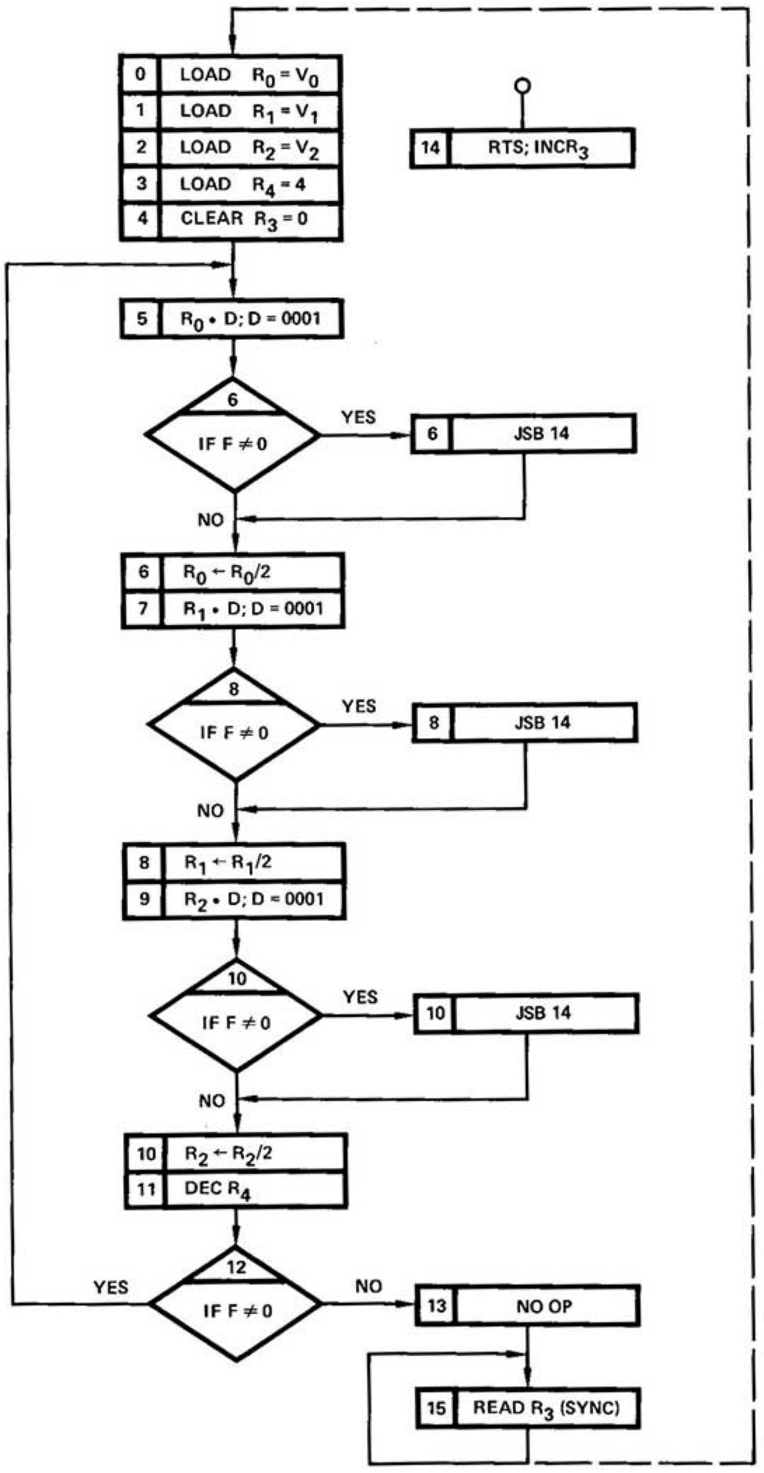


**Figure 5-4 Flowchart of Example**

**Figure 5-5 Assembly Out put Block Format**

```
ØØØØ       AM29Ø9 & AM29Ø1  ,RAMF,,DZ,,OR,,RØ & DIN H#F
ØØØ1       AM29Ø9 & AM29Ø1  ,RAMF,,DZ,,OR,,R1 & DIN 9
ØØØ2       AM29Ø9 & AM29Ø1  ,RAMF,,DZ,,OR,,R2 & DIN Ø
ØØØ3       AM29Ø9 & AM29Ø1  ,RAMF,,DZ,,OR,,R4 & DIN 4
ØØØ4       AM29Ø9 & AM29Ø1  ,RAMF,,ZB,,AND,,R3
ØØØ5 A5:   AM29Ø9 & AM29Ø1  ,,,DA,,AND,RØ,RØ & DIN 1
ØØØ6       AM29Ø9 A14,JSRFNØ & AM2901 ,RAMD,,ZB,,OR,,RØ
ØØØ7       AM29Ø9 & AM29Ø1 ,,,DA,,AND,R1,R1 & DIN 1
ØØØ8       AM29Ø9 A14,JSRFNØ & AM2901 ,RAMD,,ZB,,OR,,R1
ØØØ9       AM29Ø9 & AM29Ø1 ,,,DA,,AND,R2,R2  & DIN 1
ØØØA       AM29Ø9 A14,JSRFNØ & AM2901 ,RAMD,,ZB,,OR,,R2
ØØØB       AM29Ø9 & AM29Ø1 ,RAMF,,ZB, CNØ,SUBR,,R4
ØØØC       AM29Ø9 A5,BRFNØ & AM29Ø1
ØØØD       AM29Ø9 A15,BR & AM29Ø1
ØØØE A14:  AM29Ø9 ,RTS & AM29Ø1 ,RAMF,,ZB,CN1,ADD,,R3
ØØØF A15:  AM29Ø9 A15,BR & AM29Ø1 ,,,ZB,,OR,,R3
     END
```

```
ØØØØ XXXXØØ1ØXØ11X111 XØ11XXXXØØØØ1111
ØØØ1 XXXXØØ1ØXØ11X111 XØ11XXXXØØØ11ØØ1
ØØØ2 XXXXØØ1ØXØ11X111 XØ11XXXXØØ1ØØØØØ
ØØØ3 XXXXØØ1ØXØ11X111 XØ11XXXXØ1ØØØ1ØØ
ØØØ4 XXXXØØ1ØXØ11XØ11 X1ØØXXXXØØ11XXXX
ØØØ5 XXXXØØ1ØXØØ1X1Ø1 X1ØØØØØØØØØØØØØ1
ØØØ6 111ØØ1ØØX1Ø1XØ11 XØ11XXXXØØØØXXXX
ØØØ7 XXXXØØ1ØXØØ1X1Ø1 X1ØØØØØ1ØØØ1ØØØ1
ØØØ8 111ØØ1ØØX1Ø1XØ11 XØ11XXXXØØØ1XXXX
ØØØ9 XXXXØØ1ØXØØ1X1Ø1 X1ØØØØ1ØØØ1ØØØØ1
ØØØA 111ØØ1ØØX1Ø1XØ11 XØ11XXXXØØ1ØXXXX
ØØØB XXXXØØ1ØXØ11XØ11 ØØØ1XXXXØ1ØØXXXX
ØØØC Ø1Ø1ØØØØXØØ1XXXX XXXXXXXXXXXXXXXX
ØØØD 1111ØØØ1XØØ1XXXX XXXXXXXXXXXXXXXX
ØØØE XXXXØ11ØXØ11XØ11 1ØØØXXXXØØ11XXXX
ØØØF 1111ØØØ1XØØ1XØ11 XØ11XXXXØØ11XXXX
```

# 6 Chapter VI

## 6.1 AMPROM/80 Post Processing

When a user has completed an AMDASM assembly, the next step is to output the binary object code in a form which corresponds with the PROM's organization and/or may wish to punch the object code the program onto paper tapes to be used as input to a PROM burner.

In order to understand post processing, one must know hwo the PROMs are organized in the computer memory space.

## 6.2 PROM Organization

If, as an example, AMDASM has been executed using the command

```
–AMDASMΔP1(DEF)P2(ASM)O(PRMOUT)L(:LP:)
```

AMDASM generates binary object code of the executable statements in the file named ASM.

This binary object code is output to a file called PRMOUT.

For example, assume that the microword is 48 bits wide and the number of executable statements is 1024.

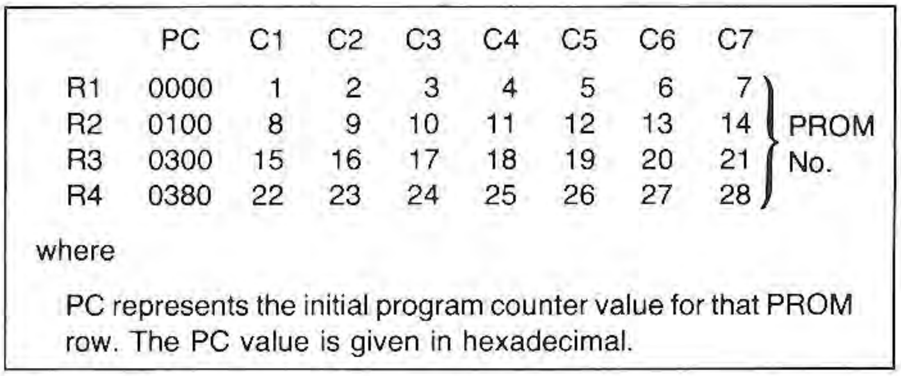
This gives a matrix 48 wide by 1024 deep as shown in Figure 6-1.

```
   Bit No.       1 2 3 4 • • • • • • • • • • • • • • • • • • • • • • • • 48

Executable   1
Instruction  2
Number       3
             4
             •
             •
             •
             1024
```

**Figure 6-1 Bit Matrix**

After PROM width and depth are specified, the Bit Matrix is subdivided to yield a PROM Map where each PROM is n bits wide by m bits deep. IF we assume that the program origin is zero for our example, the actual PROM MAP printed might appear as shown in Figure 6-2.



| | PC | C1 | C2 | C3 | C4 | C5 | C6 | C7 | |
|---|---|---|---|---|---|---|---|---|---|
| R1 | 0000 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | PROM No. |
| R2 | 0100 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | |
| R3 | 0300 | 15 | 16 | 17 | 18 | 19 | 20 | 21 | |
| R4 | 0380 | 22 | 23 | 24 | 25 | 26 | 27 | 28 | |

where

PC represents the initial program counter value for that PROM row. The PC value is given in hexadecimal.

**Figure 6-2 Sample PROM Map**

For example, PROMs shall be organized as shown in Figure 6-3.

Each executable instruction naturally has a program counter associated with it by virtue of its position in the program and/or the origin(s) that were set during the assembly execution.

This breakup of the matrix is now called a PROM map, which has associated with it not only the PROMs shown, but rows and columns as shown in Figure 6-3. We can now refer to PROM 19 by using the digits 19, or by referencing R3 for Row 3 and C5 for Column 5.

As shown in Figure 6-4, all PROMs in Row 1 are 256 (instructions) deep, but PROMs 1, 3, 5, and 6 are only 4 bits wide, while PROMs 2 and 7 are 8 bits wide and PROM 4 is 16 bits wide.

In Row 2, all PROMs are 512 (instructions) deep and PROMs8, 10, 12, and 13 are 4 bits wide, PROMs 9 and 14 are 8 bits wide and PROM 11 is 16 bits wide.

Rows 3 and 4 are each 128 (instructions) deep; PROMs 15, 17, 19, 20, 22, 24, 26, and 27 are 4 bits wide; PROMs 16, 21, 23, and 28 are 8 bits wide; and PROMs 18 and 25 are 16 bits wide.

If the user requests printing (or punching) of PROM #1 he will obtain data that is 4 by 256.

If the user requests printing of Row 3, they will obtain data (i.e., the contents of PROMs 15 through 21) in the following form:

4 X 128  8 X 128  4 X 128  16 X 128  4 X 128   4 X 128  8 X 128

If the user requests printing of Column 4, they will obtain data (i.e., the contents of PROMs 4, 11, 18, and 25) that is:

16 X 256  16 X 512  16 X 128  16 X 1218

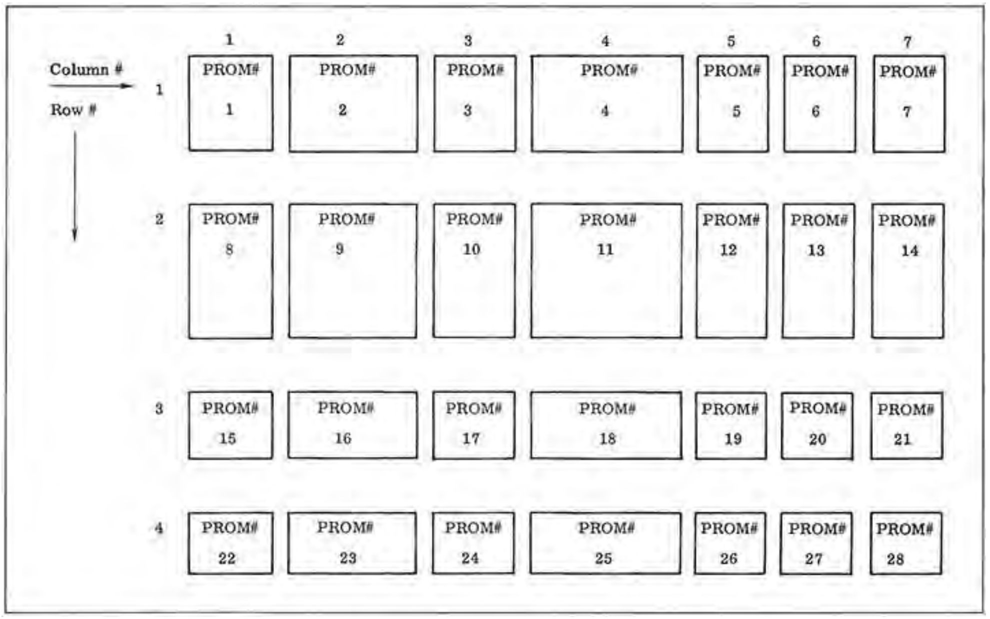


**Figure 6-3 PROM Map**

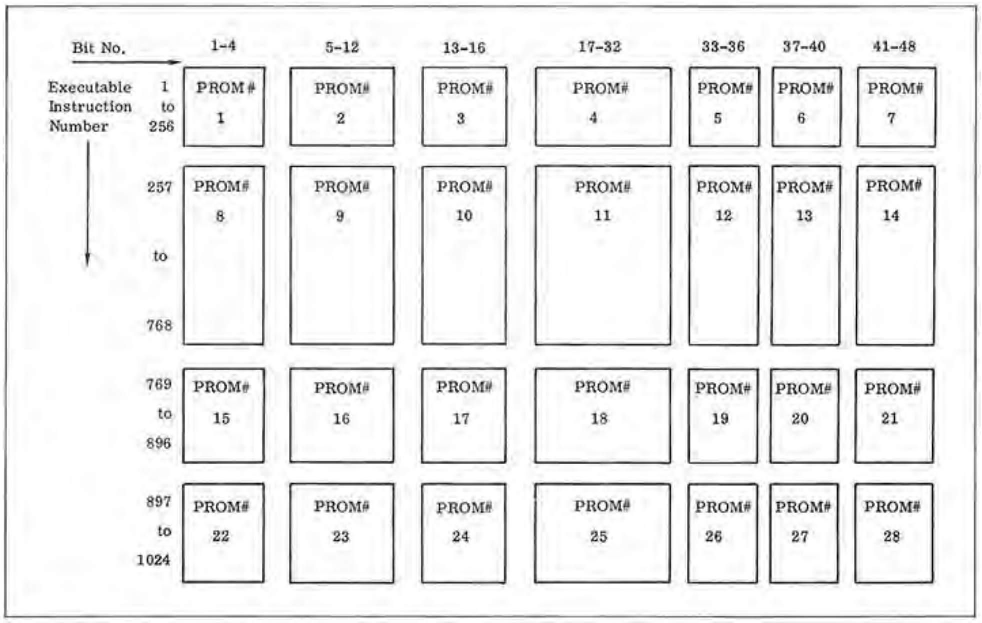


**Figure 6-4 Organization of PROMs**

## 6.3 Post Processing Features

AMPROM/80 allows the user to specify:

- The depth (number of instructions) and width (bits of the microword) for each PROM.
- Listing or suppression of listing of the PROM MAP.
- The PROMs to be punched or not punched on paper tape in BNFF or hexadecimal format.
- Listing or suppression of listing of PROM contents.
- Listing of the PROM contents by PROM rows or PROM columns or by PROM number.
- Optional automatic inversion of all bit except the “don’t care” bits.
- Specification of “don’t care” bits to be 0 or 1.

## 6.4 Execution of AMPROM

To execute AMPROM, the general form of the command is:

```
–AMPROM {Δ option}
```

To illustrate execution of AMPROM, the command

```
–AMPROMΔ PUNCH (:HP:)Δ LIST (:LP:) HEX
```

specifies the PROM MAP is to be printed, the content of the PROMs is to be printed, and the content of the PROMs is to be punched in hexadecimal.

However,

```
–AMPROMΔ NOLISTΔ NOMAPΔ PINCH (:HP:)
```

specifies that the content of PROMs is to be punched with no listing of the PROM MAP or PROM content.

Both examples assume the AMPROM input (AMDASM output) is on the default file named AMDASM.OBJ.

Note that each option specified is preceded by a space, the options may be given in any order, and the full option name or the alternate name may be used.

**Table 6-1 AMPROM/80 Options**

| OPTION | ALTERNATE OPTION | DEFAULT | MEANING |
|---|---|---|---|
| MAP | M | MAP | Print the PROM map. |
| NOMAP | NM | | Suppress printing the PROM map. If NOMAP is not specified, the program automatically prints the PROM map. |
| HEX | H | HEX | Punch the PROM output in hexadecimal format. |
| BNPF | B | | Punch the PROM output in BNPF format. If BNPF is not specified the output is automatically punched in hexadecimal |
| INVERT | I | No Inversion | If INVERT is specified, all ones are inverted to zeros, and zeros to ones, except for bits specified as "don't cares". If INVERT is not specified there is no modification of the binary object code. |
| OBJECT (filename) | O | (AMDASM.OBJ) | Specifies the name of the file on which the AMDASM object code is located. If not specified, a default file named AMDASM.OBJ is assumed. |
| PUNCH (filename) | P | (AMPROM.OUT) | Specifies the name of the file or device where punch data is to be output. If not specified the output goes to the file with the default name AMPROM.OUT. |
| NOPUNCH | NP | | Suppresses punching the PROM contents. If not specified, defaults to PUNCH. |
| LIST (filename) | L | (AMPROM.LST) | Specifies the name of the output file or device where the AMPROM output listing is to be placed. If not specified, the output automatically goes to the default file named AMPROM.LST. |
| NOLIST | NL | | Specifies that the output is not to be listed. This would be used when only punching of the output is desired. If not specified the program defaults to LIST using the default filename AMPROM.LST. |

## 6.5 APROM Filenames

AS part of –AMPROMΔ {options} the user may need to specify filename information. Whether filename information is needed will depend on whether or not a filename was specified for the binary output when AMDASM was executed [OBJECT (filename)] and whether the user wishes to receive his output at a printer, console or punched on paper tape.

Possible options are shown in Table 6-1.

If, for example, the user executed AMDASM with the command:

```
—AMDASMΔ PHASE1(DEF) PHASE2(ASM) OBJECT (PRMOUT)
```

the binary output file is stored on a file called PRMOUT.

In this case when executing AMPROM, PRMOUT must be given as the input filename.

```
—AMDASMΔ OBJECT (PRMOUT)
```

and since no LIST or PUNCH is specified, all output will be to the default filenames.

If the user executes AMDASM with the command

```
—AMDASMΔ PHASE 1 (DEF) PHASE 2 (ASM)
```

the binary object code is output to the default file called AMDASM.OBJ.

AMPROM assumes the binary object code is stored on a file named AMDASM.OBJ if no input filename is given (i.e., the AMPROM input default filename is AMDASM.OBJ). Thus, in this example, the command

```
–AMPROM
```

will cause AMPROM to be executed and specifies (by default) that the binary object code is to be put from a file named AMDASM.OBJ.

The command

```
–AMPROMΔ NOLISTΔ PUNCH (PRMPNC) OBJECT (PRMDAT)
```

specifies that listing of the PROM content is to be suppressed, the output for punching a paper tape is to be written on a file called PRMPNC, and the input (binary object code) for execution of AMPROM is to be from a file called PRMDAT.

This assumes the AMDASM output was stored on a file called PRMDAT.

Note that the options may be given in any order desired by the user.

## 6.6 Required AMPROM Input

Once AMPROM has begun execution the user will be acting interactively with the console. The user will receive messages from the console and will be expected to input resources followed by a carriage return. The terminal will print the requested output of a message requesting additional input. When execution is complete, control returns to ISIS$^{©}$.

A sample of the console messages is given below. For this example, underlined letters are used to illustrate the user's input. Following the example is a table of the substitutes that may be used for these letters.

To begin execution the user has input –APROM. The terminal responds by printing:

```
DON'T CARES? Z
```

For example:

```
DON'T CARES?  1
ENTER PROM WIDTH(S)   W
```

For example:

```
ENTER PROM WIDTH(S)  4*8, 4
ENTER PROM DEPTH(S)  D
```

For example:

```
ENTER PROM DEPTH(S)  128
```

Is a MAP listing at the output device is requested the PROM map is output here. Then the console prints

```
WHICH PROMS DO YOU WISH TO PRINT?  Q
```

For example:

```
WHICH PROMS DO YOU WISH TO PRINT?  5-7
```

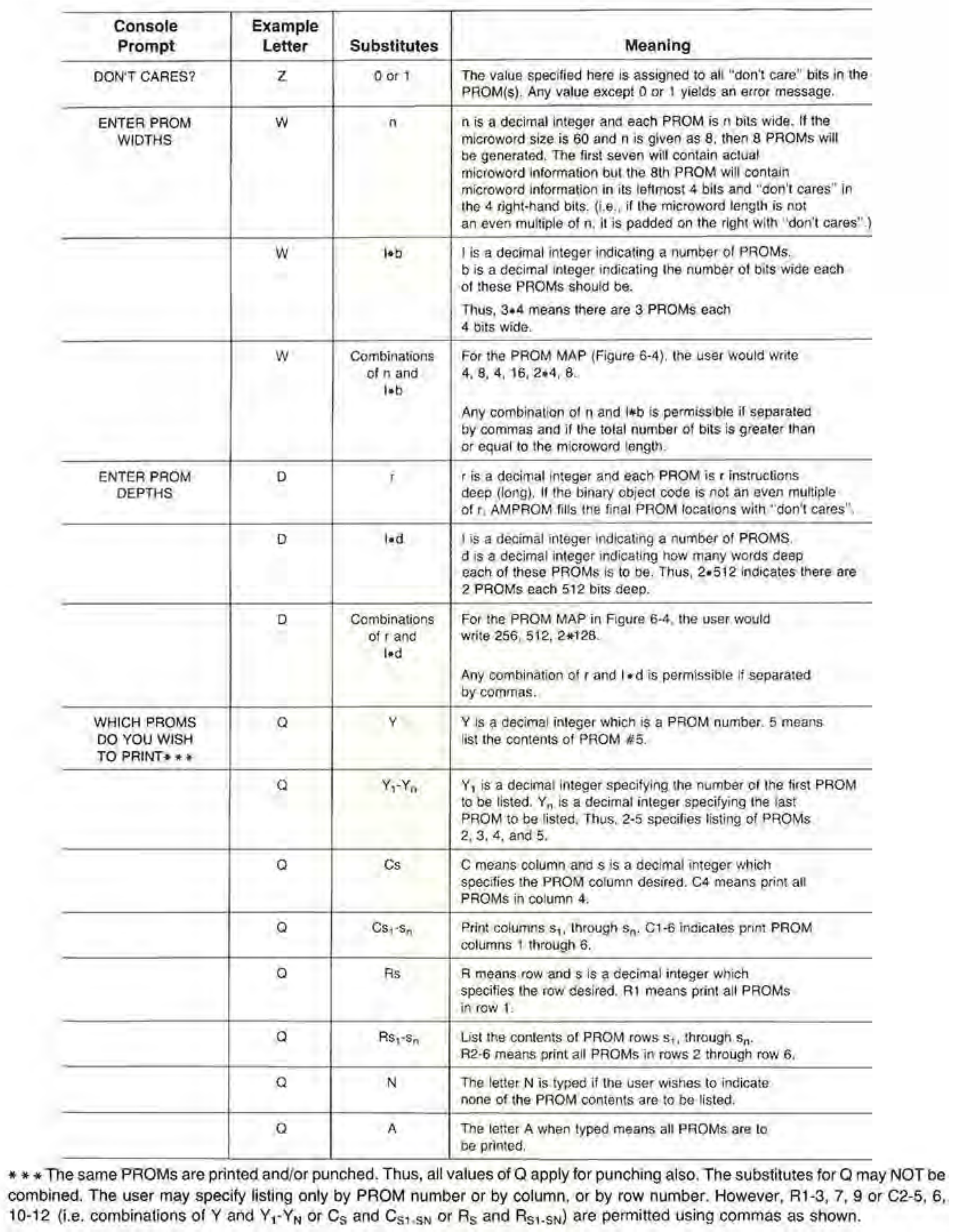
If printing of the PROM content was specified, the PROM content is printed here. These same PROMs will be punched unless NOPUNCH was specified. The punch device should be turned on before keying in the PROMs to be printed and punched.

When paper tape punching is completed, control is returned to ISIS$^{©}$.

## 6.7 *Input Substitutes Permitted*

When the terminal requests information, the substitutes permitted for the letters used in the example are shown in Table 6-2.

**Table 6-2 AMPROM/80 Input Substitutes**



| Console Prompt | Example Letter | Substitutes | Meaning |
|---|---|---|---|
| DON'T CARES? | Z | 0 or 1 | The value specified here is assigned to all "don't care" bits in the PROM(s). Any value except 0 or 1 yields an error message. |
| ENTER PROM WIDTHS | W | n | n is a decimal integer and each PROM is n bits wide. If the microword size is 60 and n is given as 8, then 8 PROMs will be generated. The first seven will contain actual microword information but the 8th PROM will contain microword information in its leftmost 4 bits and "don't cares" in the 4 right-hand bits. (i.e., if the microword length is not an even multiple of n, it is padded on the right with "don't cares".) |
| | W | I•b | I is a decimal integer indicating a number of PROMs. b is a decimal integer indicating the number of bits wide each of these PROMs should be. Thus, 3•4 means there are 3 PROMs each 4 bits wide. |
| | W | Combinations of n and I•b | For the PROM MAP (Figure 6-4), the user would write 4, 8, 4, 16, 2•4, 8. Any combination of n and I*b is permissible if separated by commas and if the total number of bits is greater than or equal to the microword length. |
| ENTER PROM DEPTHS | D | r | r is a decimal integer and each PROM is r instructions deep (long). If the binary object code is not an even multiple of r, AMPROM fills the final PROM locations with "don't cares". |
| | D | I•d | I is a decimal integer indicating a number of PROMS. d is a decimal integer indicating how many words deep each of these PROMs is to be. Thus, 2•512 indicates there are 2 PROMs each 512 bits deep. |
| | D | Combinations of r and I•d | For the PROM MAP in Figure 6-4, the user would write 256, 512, 2*128. Any combination of r and I•d is permissible if separated by commas. |
| WHICH PROMS DO YOU WISH TO PRINT*** | Q | Y | Y is a decimal integer which is a PROM number. 5 means list the contents of PROM #5. |
| | Q | $Y_1$-$Y_n$ | $Y_1$ is a decimal integer specifying the number of the first PROM to be listed. $Y_n$ is a decimal integer specifying the last PROM to be listed. Thus, 2-5 specifies listing of PROMs 2, 3, 4, and 5. |
| | Q | Cs | C means column and s is a decimal integer which specifies the PROM column desired. C4 means print all PROMs in column 4. |
| | Q | $Cs_1$-$s_n$ | Print columns $s_1$, through $s_n$. C1-6 indicates print PROM columns 1 through 6. |
| | Q | Rs | R means row and s is a decimal integer which specifies the row desired. R1 means print all PROMs in row 1. |
| | Q | $Rs_1$-$s_n$ | List the contents of PROM rows $s_1$, through $s_n$. R2-6 means print all PROMs in rows 2 through row 6. |
| | Q | N | The letter N is typed if the user wishes to indicate none of the PROM contents are to be listed. |
| | Q | A | The letter A when typed means all PROMs are to be printed. |

*** The same PROMs are printed and/or punched. Thus, all values of Q apply for punching also. The substitutes for Q may NOT be combined. The user may specify listing only by PROM number or by column, or by row number. However, R1-3, 7, 9 or C2-5, 6, 10-12 (i.e. combinations of Y and $Y_1$-$Y_N$ or $C_S$ and $C_{S1\text{-}SN}$ or $R_S$ and $R_{S1\text{-}SN}$) are permitted using commas as shown.

## 6.8 BNPF Paper Tape Option

When BNPF is specified as an option, the tape is punched in the BNPF format. B is punched as the first character, then a P (for a one) or an N (for a zero) is punched for each bit in the width of this PROM, then an F is punched as the last character for this row of PROM data. This continues until all rows (the depth) of the PROM are punched.

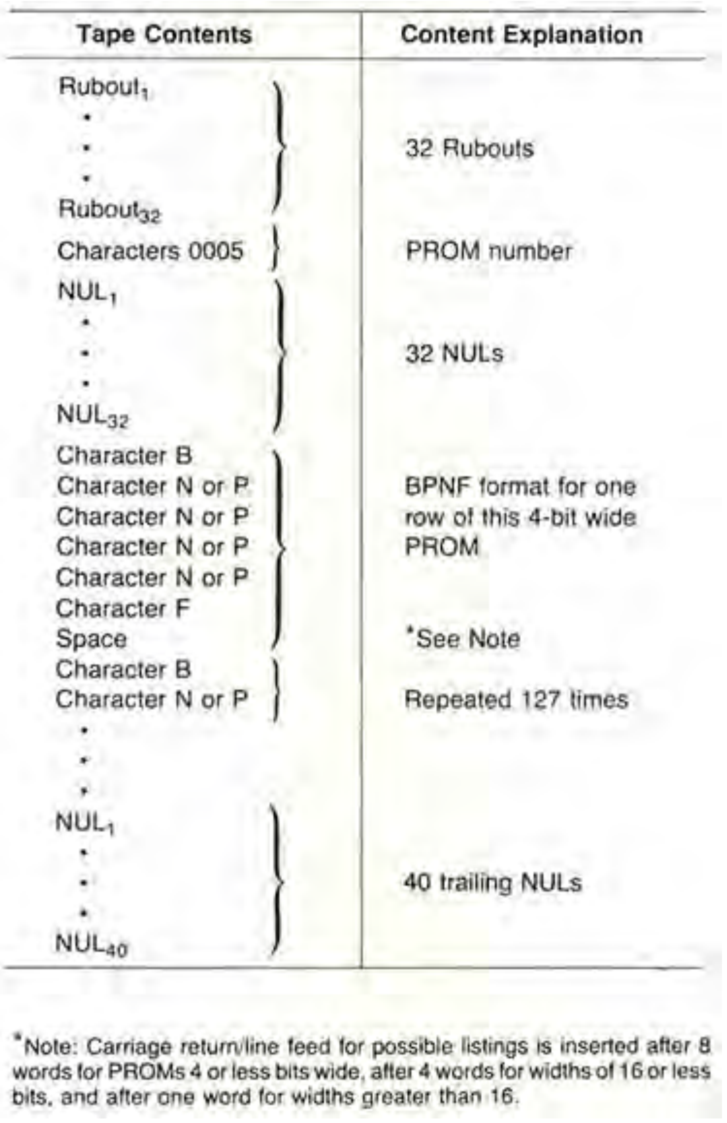
Before the first BNPF for each PROM is punched, the program punches identification on the tape which consists of:

- 32 Rubouts
- 4 ASCII characters which are the PROM number
- 32 NULs to be used as the leader when loading the PROM burner tape reader

After the PROM data is punched, 40 NULS are punched to facilitate tape handling.

For example, if PROM#5 is 4 bits wide by 128 bits deep, and begins at origin zero, the paper tape will appear as shown in Table 6-3.

**Table 6-3 BNPF Paper Tape Contents**



| Tape Contents | Content Explanation |
|---|---|
| $\text{Rubout}_1$<br>•<br>•<br>•<br>$\text{Rubout}_{32}$ | 32 Rubouts |
| Characters 0005 | PROM number |
| $\text{NUL}_1$<br>•<br>•<br>•<br>$\text{NUL}_{32}$ | 32 NULs |
| Character B<br>Character N or P<br>Character N or P<br>Character N or P<br>Character N or P<br>Character F<br>Space | BPNF format for one row of this 4-bit wide PROM<br><br>*See Note |
| Character B<br>Character N or P<br>•<br>•<br>• | Repeated 127 times |
| $\text{NUL}_1$<br>•<br>•<br>•<br>$\text{NUL}_{40}$ | 40 trailing NULs |

*Note: Carriage return/line feed for possible listings is inserted after 8 words for PROMs 4 or less bits wide, after 4 words for widths of 16 or less bits, and after one word for widths greater than 16.

## 6.9 Hexadecimal Paper Tape Option

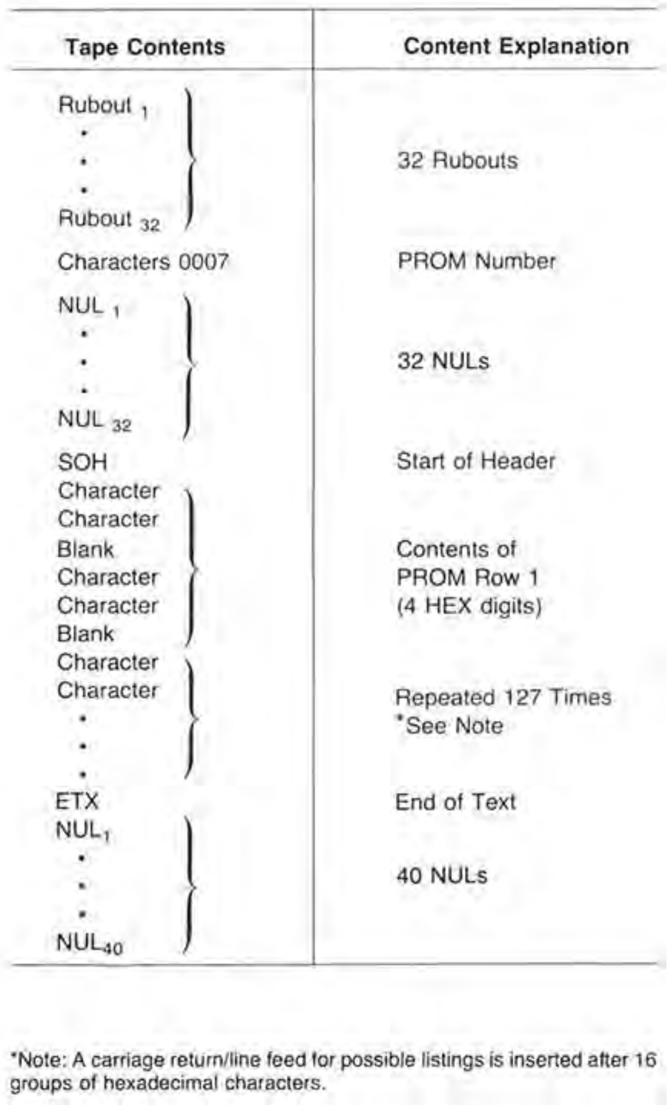
When punching is desired, and HEX is specified or assumed by default, the PROM contents are punched in the DATAS I/O hexadecimal format.

The same initial data (32 Rubouts, PROM number, and 32 NULs) is punched as is punched for the BNPF format, followed by the PROM contents in hexadecimal.

For PROMs 4 or less bits wide, one hexadecimal character and a space is punched. For PROMs greater than 4 bits wide, two hexadecimal characters and a space are punched. Thus, two characters, space, two characters, space, would be punched for either 2 rows of an 8-bit PROM, or for 1 row of a 16-bit wide PROM.

Thus, if PROM #7 (16 bits x 128 words) is punched, the output will be:

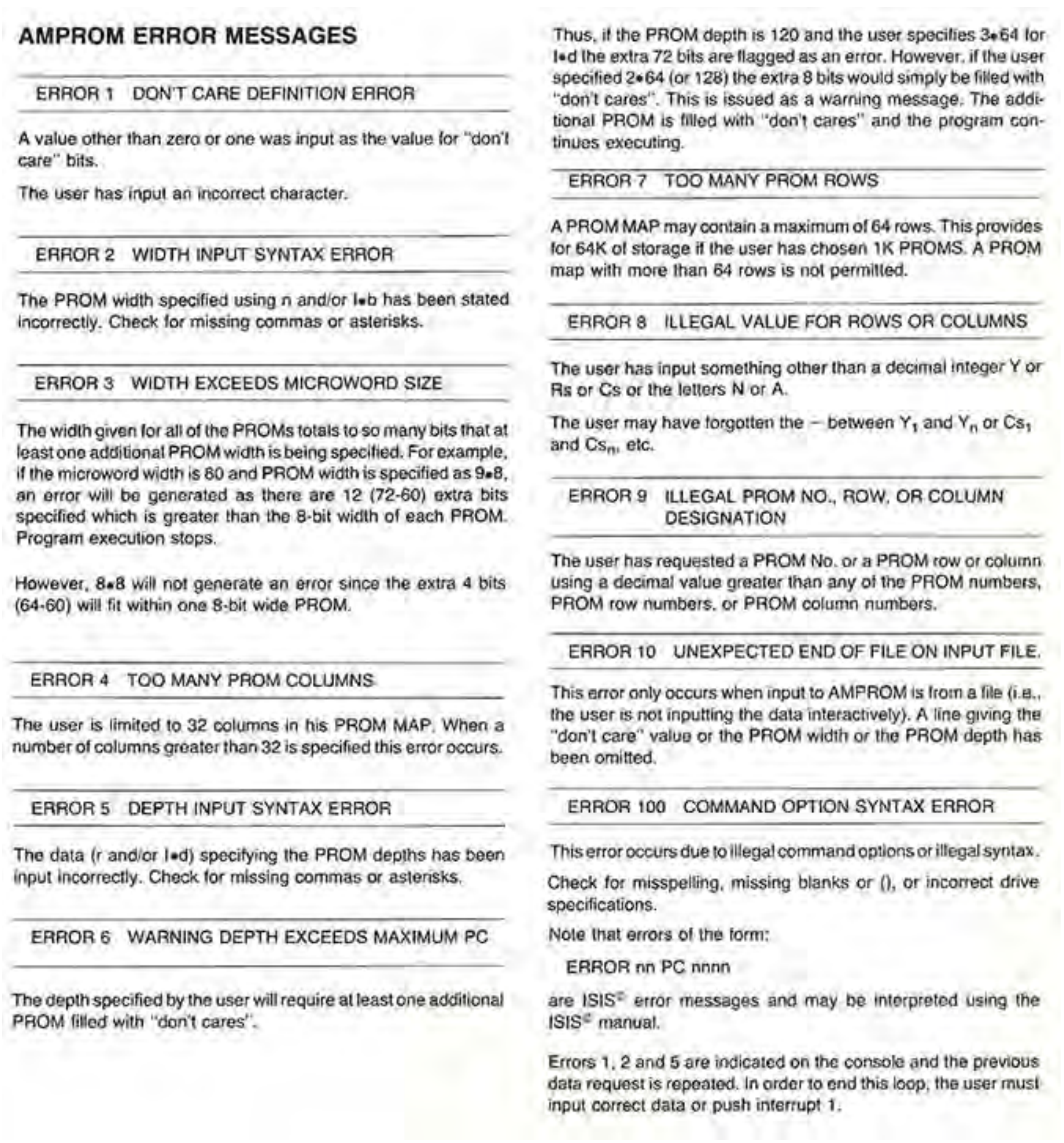
**Table 6-4 Hexadecimal Paper Tape Contents**



| Tape Contents | Content Explanation |
|---|---|
| Rubout $_1$ | |
| • | |
| • | 32 Rubouts |
| • | |
| Rubout $_{32}$ | |
| Characters 0007 | PROM Number |
| NUL $_1$ | |
| • | |
| • | 32 NULs |
| • | |
| NUL $_{32}$ | |
| SOH | Start of Header |
| Character | |
| Character | |
| Blank | Contents of |
| Character | PROM Row 1 |
| Character | (4 HEX digits) |
| Blank | |
| Character | |
| Character | Repeated 127 Times |
| • | *See Note |
| • | |
| • | |
| ETX | End of Text |
| NUL$_1$ | |
| • | |
| • | 40 NULs |
| • | |
| NUL$_{40}$ | |

*Note: A carriage return/line feed for possible listings is inserted after 16 groups of hexadecimal characters.

## 6.10 AMPROM Error Messages



### AMPROM ERROR MESSAGES

ERROR 1 DON'T CARE DEFINITION ERROR

A value other than zero or one was input as the value for "don't care" bits.

The user has input an incorrect character.

ERROR 2 WIDTH INPUT SYNTAX ERROR

The PROM width specified using *n* and/or I•b has been stated incorrectly. Check for missing commas or asterisks.

ERROR 3 WIDTH EXCEEDS MICROWORD SIZE

The width given for all of the PROMs totals to so many bits that at least one additional PROM width is being specified. For example, if the microword width is 60 and PROM width is specified as 9•8, an error will be generated as there are 12 (72-60) extra bits specified which is greater than the 8-bit width of each PROM. Program execution stops.

However, 8•8 will not generate an error since the extra 4 bits (64-60) will fit within one 8-bit wide PROM.

ERROR 4 TOO MANY PROM COLUMNS

The user is limited to 32 columns in his PROM MAP. When a number of columns greater than 32 is specified this error occurs.

ERROR 5 DEPTH INPUT SYNTAX ERROR

The data (r and/or I•d) specifying the PROM depths has been input incorrectly. Check for missing commas or asterisks.

ERROR 6 WARNING DEPTH EXCEEDS MAXIMUM PC

The depth specified by the user will require at least one additional PROM filled with "don't cares".

Thus, if the PROM depth is 120 and the user specifies 3•64 for I•d the extra 72 bits are flagged as an error. However, if the user specified 2•64 (or 128) the extra 8 bits would simply be filled with "don't cares". This is issued as a warning message. The additional PROM is filled with "don't cares" and the program continues executing.

ERROR 7 TOO MANY PROM ROWS

A PROM MAP may contain a maximum of 64 rows. This provides for 64K of storage if the user has chosen 1K PROMS. A PROM map with more than 64 rows is not permitted.

ERROR 8 ILLEGAL VALUE FOR ROWS OR COLUMNS

The user has input something other than a decimal integer Y or Rs or Cs or the letters N or A.

The user may have forgotten the – between $Y_1$ and $Y_n$ or $Cs_1$ and $Cs_n$, etc.

ERROR 9 ILLEGAL PROM NO., ROW, OR COLUMN DESIGNATION

The user has requested a PROM No. or a PROM row or column using a decimal value greater than any of the PROM numbers, PROM row numbers, or PROM column numbers.

ERROR 10 UNEXPECTED END OF FILE ON INPUT FILE.

This error only occurs when input to AMPROM is from a file (i.e., the user is not inputting the data interactively). A line giving the "don't care" value or the PROM width or the PROM depth has been omitted.

ERROR 100 COMMAND OPTION SYNTAX ERROR

This error occurs due to illegal command options or illegal syntax.

Check for misspelling, missing blanks or (), or incorrect drive specifications.

Note that errors of the form:

ERROR nn PC nnnn

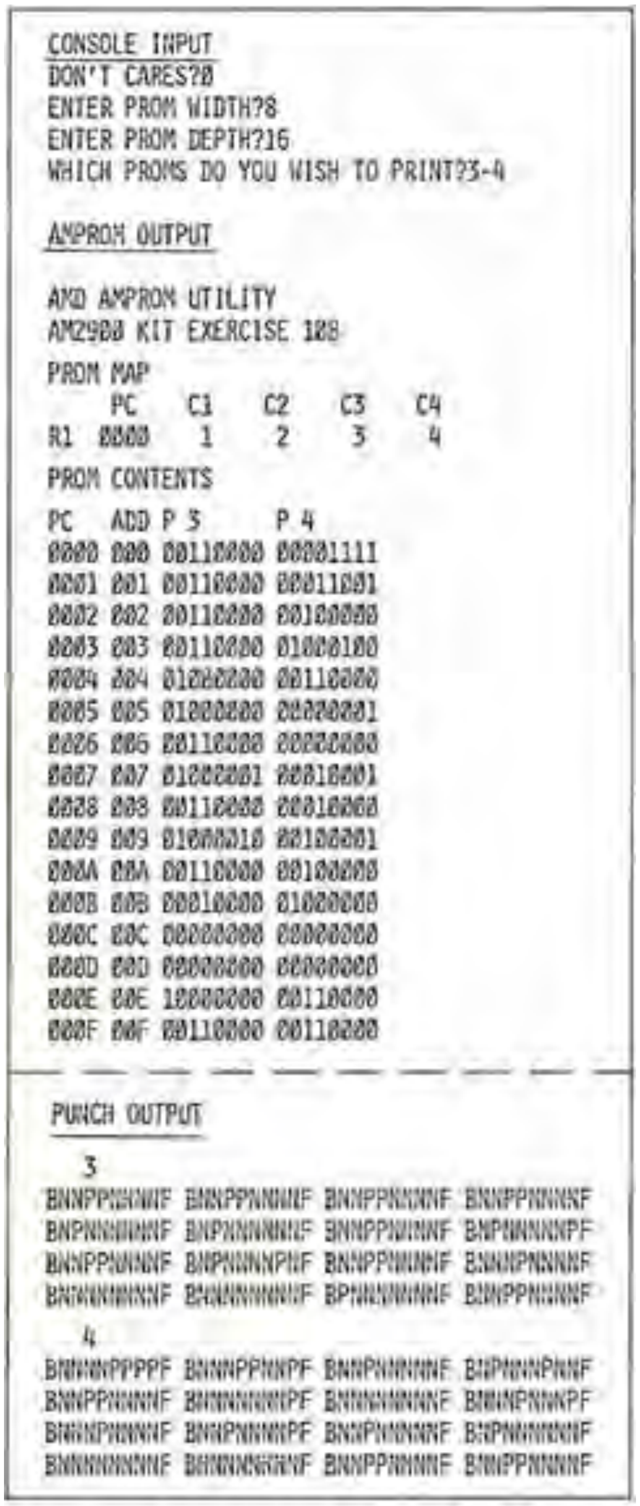
are ISIS© error messages and may be interpreted using the ISIS© manual.

Errors 1, 2 and 5 are indicated on the console and the previous data request is repeated. In order to end this loop, the user must input correct data or push interrupt 1.

# 7 Chapter VII

## 7.1 Example of AMPROM/80

Figure 7-1 is an example of AMPROM/80 for the AMD 2900 Learning & Evaluation kit.



```
CONSOLE INPUT
DON'T CARES?0
ENTER PROM WIDTH?8
ENTER PROM DEPTH?16
WHICH PROMS DO YOU WISH TO PRINT?3-4

AMPROM OUTPUT

AMD AMPROM UTILITY
AM2900 KIT EXERCISE 100
PROM MAP
       PC    C1    C2    C3    C4
R1  0000     1     2     3     4
PROM CONTENTS
PC   ADD P 3      P 4
0000 000 00110000 00001111
0001 001 00110000 00011001
0002 002 00110000 00100000
0003 003 00110000 01000100
0004 004 01000000 00110000
0005 005 01000000 00000001
0006 006 00110000 00000000
0007 007 01000001 00010001
0008 008 00110000 00010000
0009 009 01000010 00100001
000A 00A 00110000 00100000
000B 00B 00010000 01000000
000C 00C 00000000 00000000
000D 00D 00000000 00000000
000E 00E 10000000 00110000
000F 00F 00110000 00110000
```

```
PUNCH OUTPUT

  3
BNNPPNNNNF BNNPPNNNNF BNNPPNNNNF BNNPPNNNNF
BNPNNNNNNF BNPNNNNNNF BNNPPNNNNF BNPNNNNNPF
BNNPPNNNNF BNPNNNNPNF BNNPPNNNNF BNNNPNNNNF
BNNNNNNNNF BNNNNNNNNF BPNNNNNNNF BNNPPNNNNF
  4
BNNNNPPPPF BNNNPPNNPF BNNPNNNNNF BNPNNNPNNF
BNNPPNNNNF BNNNNNNNPF BNNNNNNNNF BNNNPNNNPF
BNNNPNNNNF BNNPNNNNPF BNNPNNNNNF BNPNNNNNNF
BNNNNNNNNF BNNNNNNNNF BNNPPNNNNF BNNPPNNNNF
```

Advanced Micro Devices
901 Thompson Place
Sunnyvale, Calif. 94086
Telephone: (408) 732-2400
TWX: 910-339-9280
Telex: 34-6306